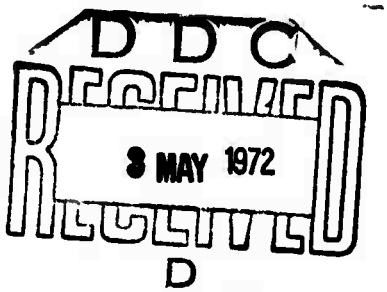
Computer Corporation of America

575 Technology Square
Cambridge
Massachusetts 02139

617-491-3670

Semi-Annual Technical Report

March 1972



The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U.S. Government.

Approved for public release; distribution unlimited.

ARPA Order Number	1731
Program Code Number	61101D
Name of Contractor	Computer Corporation of America
Effective Date of Contract	8 January 1971
Contract Expiration Date	7 January 1973
Contract Number	DAHC04-71-C-0011
Principal Investigator and Phone Number	Thomas Marill, 617-491-3670
Project Scientist and Phone Number	Kenneth E. Curewitz, 617-491-3670
Short Title of Work	Network Data Handling System (Datacomputer Project)
Amount of Contract	$1,509,256 (including fee)

Approved for public release; distribution unlimited.

Sponsored by

Advanced Research Projects Agency

ARPA Order No. 1731

Table of Contents

Technical Report Summary

The purpose of the project is to advance that technology associated with data handling in computer networks. The components of this study project are: the design and implementation of a network data language which will provide to network participants a uniform and convenient means of access to remote data; the investigation of using a large capacity storage device (in the trillion bit range), for network data storage; and the specification and analysis of use of a specialized datacomputer to provide data handling facilities for remote and local network users.

The project plan is to identify uses to be made of the datacomputer through formal and informal contacts and then to define datalanguage so that it will accommodate the identified application requirements. Hardware selected as components of the datacomputer must provide the capacity and capability required to do the datacomputer job--that is, the hardware must satisfy the storage and processing requirements imposed by the anticipated uses of the datacomputer on the network. The next steps are the hardware procurement and design phases which precede the actual implementation and testing stages of this work. By virtue of the modular system design it is expected that these last few steps will be overlapped for several of the hardware components. As an example, it is planned to have a basic datacomputer capability running with disks as the mass storage device, prior to integrating the UNICON Laser Mass Memory System as the tertiary store. Subsequent to the implementation and integration phases for the various datacomputer modules it is planned to subject the system to on-line user applications and to "fine tune" the system based on the results of this experience. The final system will be analyzed and evaluated to provide the project results information.

With regard to technical results, the main body of the system design has been completed and is contained in this report in the form of Working Papers Numbered 3 and 4, entitled respectively "Datalanguage" and "Timing and Storage Estimates".

Two significant datacomputer hardware components have been acquired for the project. First, the Digital Equipment Corporation's PDP-10 computer has been delivered to CCA in Cambridge and has successfully undergone acceptance testing. Second, the Precision Instrument Company's UNICON 690/212 Laser Mass Memory System has been delivered to the NASA-Ames facility at Moffett Field. This system was to have been readied for acceptance testing and shipment to CCA in Cambridge to be integrated into the datacomputer. It was decided to keep the UNICON at NASA-Ames in order to assure the timely availability of the manufacturer's design and development personnel.

The implications of the current work substantiate the viability of the original statement of work. The method developed for communicating with the datacomputer, that is, datalanguage (as specified in Working Paper No. 3) and the timing and storage values which the system will exhibit (Working Paper No. 4) combine to reinforce the concept of the datacomputer as an important resource for network data storage.

A great deal of work remains to be completed on this project--to prove that the system can be assembled as proposed, and to improve the system based upon the requirements of real time operations. The datacomputer in operation will provide to network technology and to computer operations technology a new, fundamental capability for exploration and improvement that should yield important, future results. Typical of these capabilities is the on-line (immediate) availability of extremely large files which will be maintained and may be accessed simultaneously by a number of remote and local users, and the ability to subset quickly and retrieve from these files only that information which is relevant to a particular application.

Datacomputer Project
Working Paper No. 3
October 29, 1971

Contract No. DAHC04-71-C-0011
ARPA Order 1731

Computer Corporation of America
575 Technology Square
Cambridge, Massachusetts 02139

Preface

The present document discusses the datalanguage of the datacomputer system. This language is the notation for interacting with the datacomputer and defines the capabilities of that system.

Other documents--issued and to be issued in the present series--discuss the software architecture of the datacomputer, the hardware configuration, and related topics. All documents in the series are working papers, and subject to revision without notice.

Table of Contents

Chapter 1
Introduction to Datalanguage

1.1 The Datacomputer

The dataco puter is a system which performs data storage and data management functions.

One may consider the datacomputer as a black box with multiple physical ports to which processors can be interfaced. (See Fig. 1.1)

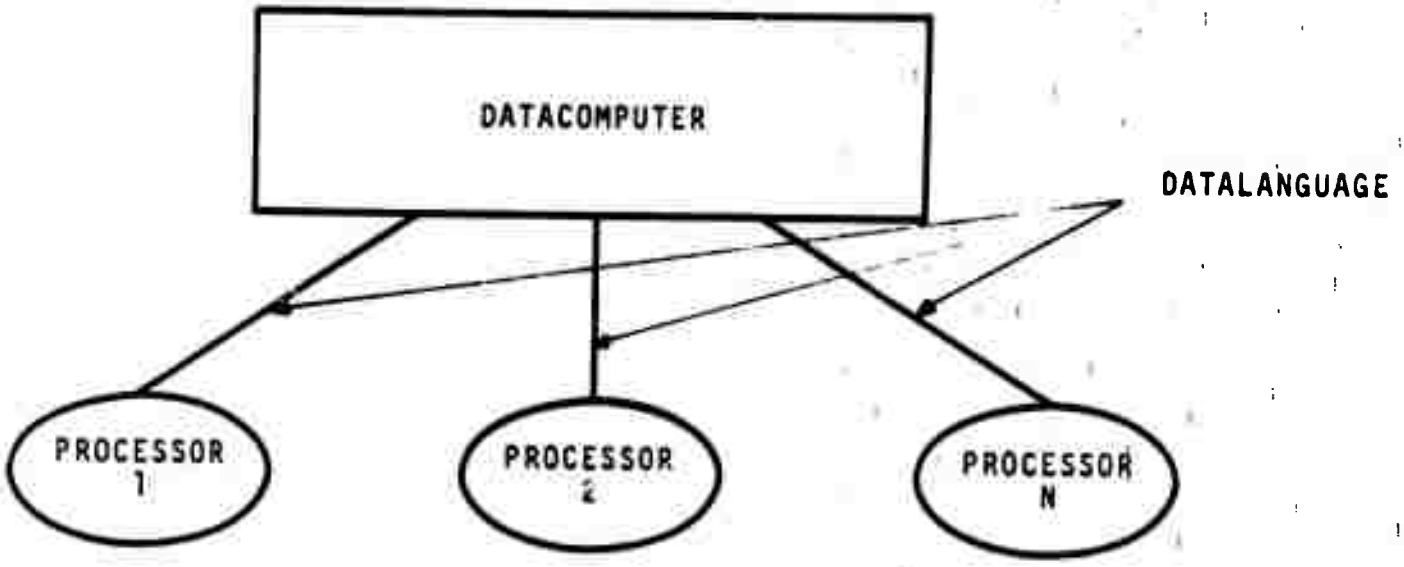


Figure 1.1

Each of the processors can itself have multiple users, which can avail themselves of the data storage and data management services that the black box offers.

The datacomputer offers the following types of service:

1. On-line storage of files and file descriptions. These files can be extremely large, with an upper limit of 10^{12} bits.

2. Retrieval of data (whole files, subsets of files, individual data elements).

3. File maintenance functions, that is, addition of new data, deletion of old data, changes to existing data.

4. Data reformatting.

5. Backup and recovery mechanisms.

6. Accounting.

7. Data security, preventing users from getting unauthorized access to data.

8. Data sharing, allowing multiple users to access the same data bases.

Interaction with the datacomputer is through a specialized system of notation called datalanguage. Within datalanguage, one can express all requests for data storage and data management services of which the datacomputer is capable. When datalanguage statements are presented at the ports of the datacomputer, the system proceeds to perform the desired service request. Datalanguage statements can originate in a user program directly, in the compilation process of a user program, or in the operating system of the external computers.

1.2 Presentation of Datalanguage to the Datacomputer

The datacomputer for the ARPANET is physically connected to an IMP and the Illiac IV, as shown below:

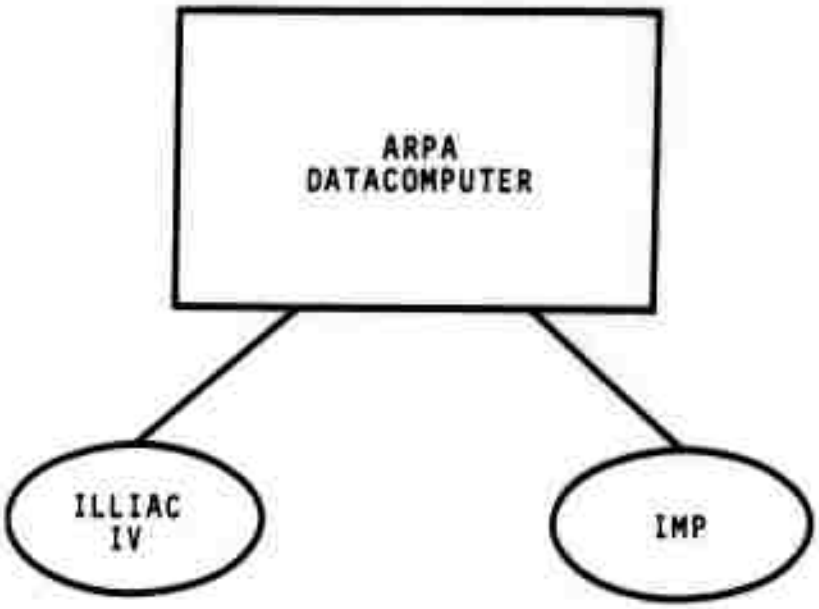


Figure 1.2

The datacomputer sees all users as remote programs connected to it via one or more one-way logical paths. Each using program establishes one path for the communication of datalanguage to the datacomputer. Additional paths may be established for the transfer of data in either direction. Thus, regardless of their means of physical connection, all using programs appear to the datacomputer to fit the model below:

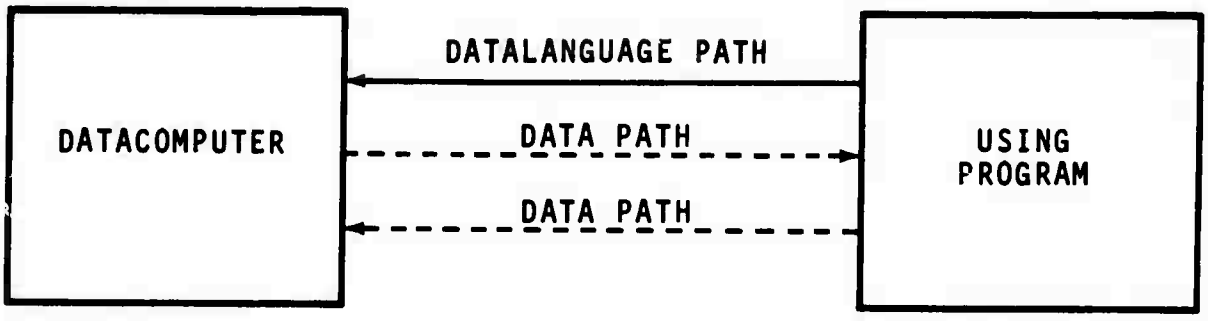


Figure 1.3

The datacomputer end of each path is a datacomputer logical port. Once the logical paths are established, the operation proceeds as follows:

1. Datalanguage statements are transmitted along the datalanguage path to the datacomputer.

2. When a complete statement or block of statements is received by the datacomputer, it is interpreted.

3. As a result of interpretation, one or more of the following happens:

   a. the datacomputer accepts data at one or more logical ports
   b. the datacomputer outputs data at one or more logical ports
   c. the datacomputer performs internal data management functions.

This paper is concerned with the specification of the datalanguage statements alluded to in (1) above.

1.3 Datalanguage Examples

Consider a file of experimental weather observations stored at the datacomputer. In a very simple file all the observations have the same format:

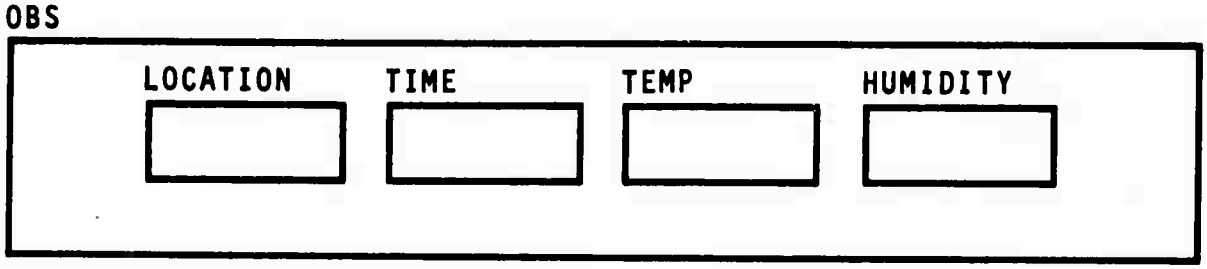


Figure 1.4

Each observation contains location, time, temperature, and humidity. The file of identically formatted weather observations is named WEATHER, and can be pictured thus:

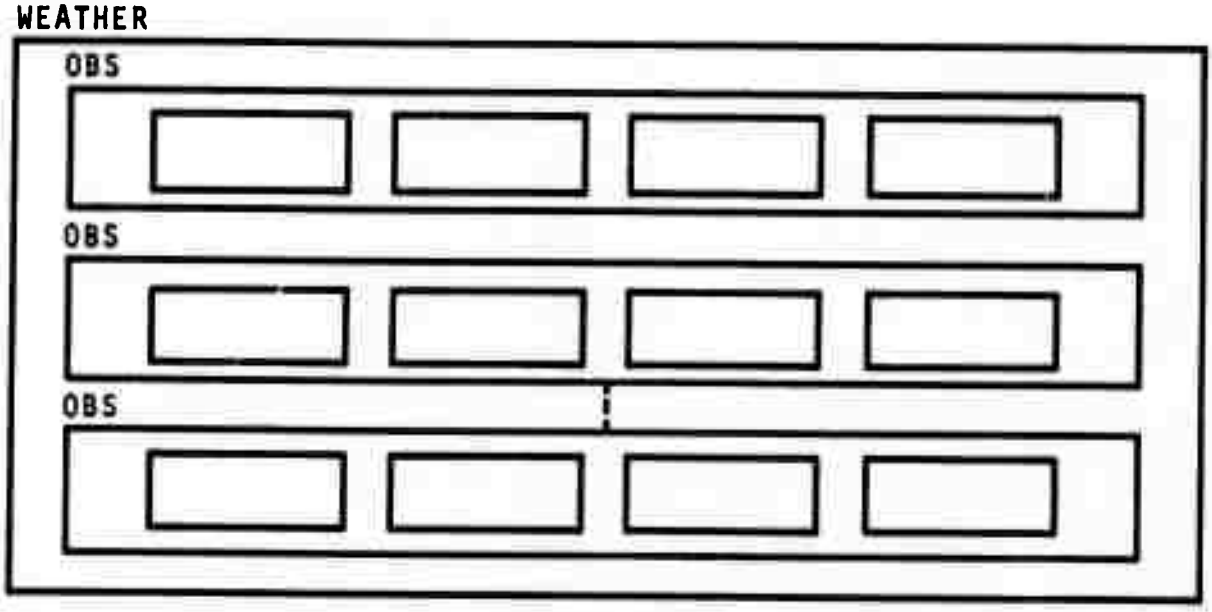


Figure 1.5

A program wanting to receive a copy of this file from the datacomputer establishes a datalanguage path and a single data path.

This data path is a one-way path from the datacomputer to the program. The datacomputer logical port to this path is given the reference name X. Then the following datalanguage statements are transmitted on the datalanguage path:

```
OPEN CCA.RAW.CLIM.WEATHER.....
TRANS X,WEATHER
```

The first statement opens the file. The open operation establishes the user's right to access the file and makes a file description and map available to the datalanguage compiler. The external file name, which is used in the open statement, is a hierarchical name, so that it is unique among the large number of file names known to the datacomputer. Once the file is open, the external file name need not be used. The file description contains an internal name for the file, which is used in the second datalanguage statement. This statement causes the contents of the file to be transmitted through port X.

The ellipsis (...) following the file name in the open statement is not part of the language. It is present here to indicate that additional information, such as a password or class of usage might be supplied as part of the open. From the datacomputer's point of view, this is a fairly simple request. The file is retrieved exactly as stored, with no reformatting and no selection of data elements. Even the formatting of the output stream is defaulted -- successive observations are delimited just as they are in the stored file. If this is the only type of request that is ever to be input for the WEATHER file, it can be described to the datacomputer as a single string of bits. However, so that we can go into more interesting examples, let us assume the datacomputer is aware of the structure of the file as pictured on the preceding page. It also knows the parts of the file by the names used in the diagrams, and knows the format of data inside each box.

Now consider a using program that is interested only in the observations for the city of Boston. For convenience, all our examples will name the output port (output from the point of view of the datacomputer) with the name X. This program sends the statements:

```
OPEN CCA.RAW.CLIM.WEATHER,...
TRANS X,OBS WITH LOCATION = 'BOSTON'
```

This time the set of observations with a location of Boston is transmitted through port X. The TRANS statement takes a data-language expression for its second argument. Besides the operators WITH and =, which appear in this example, such an expression may include the operators ← (assign), ADD, DEL, AND, OR, NOT, WITHOUT, ¬=, <, >, ¬<, ¬>, enclosure, subscripting, function call, and a small group of computational operators. The value of such an expression (i.e., that which you get by evaluating it) is a set of data items. All the operators named are defined to operate on sets. A few examples of datalanguage expressions that might be used in the TRANS statement above:

```
OBS WITH LOCATION = 'CLEVELAND' OR TEMP = 17
OBS WITH LOCATION  = 'NEW YORK' AND HUMIDITY >90
OBS WITH TEMP <HUMIDITY
OBS WITH TEMP = F(TIME)
```

In the last case, F is a function defined by a user in datalanguage. F looks up the time in a table and returns the table entry for comparison to the temperature. The effect is to select the observations having certain (time, temperature) pairs.

Another program might want only part of each observation transmitted, instead of the entire observation. In this case, the desired format is specified in a data description and associated with the output port. Such a description might indicate that items called OBS were to contain only a LOCATION and a TEMP. These are identical to the stored OBSs, but have no TIME and no HUMIDITY. The new description is associated with output port X, and then the following datalanguage statements are input to the datacomputer:

```
OPEN CCA.RAW.CLIM.WEATHER...
X.OBS ← WEATHER.OBS
```

The left-pointing arrow indicates the operation of assignment. Assignment is the transferrence of data from the item on the right of the arrow to the item on the left. In this case, data is transferred from each OBS in the file to an OBS associated with the output port. After each output OBS is assigned values, it is transmitted. The reformatting occurs automatically as part of a matching process. In this process, components of X.OBS and WEATHER.OBS are matched by name, WEATHER.OBS.TEMP is assigned to X.OBS.TEMP, and WEATHER.OBS.LOCATION is assigned to X.OBS.LOCATION.

The description of the output OBS can specify that LOCATION be encoded in a different character set than the LOCATION in the file. It can use a different format for the value of TEMP. TEMP might be a 32-bit integer in the file, and a 36-bit integer in the output. TEMP might be an IBM360 floating point number in the file, and a PDP-10 floating point number in the output. The fields may be fixed-length in the file and variable-length in the output. In general, a large number of low-level formatting options (designed to accomodate a variety of machines) are specifiable in the data description, and automatically compensated for when they differ in the operands of a datalanguage expression.

Another program wishes to add observations to the file. It sets up a data path to the datacomputer with port name Y, and sends the following datalanguage:

```
OPEN CCA.RAW.CLIM.WEATHER,...
TRANS Y,ADD OBS
```

The second statement causes a set of observations to be accepted by the datacomputer at port Y and added to the WEATHER file. These observations are in exactly the format specified in the file description. They are added to the file by:

a. appending them to the data that is present, if the file is not ordered by content, or

b. inserting them in sequence if the file description specifies that the file is ordered by content.

As in the previous example, the second argument to the TRANS statement is an expression. The direction of the transmission is derived from the port name (since all ports are one-way). The use of an expression may seem puzzling until it is realized that expressions evaluate to sets of data items. This expression evaluates to a set of OBSs that are part of the WEATHER file. Specifically, they are the OBSs that are being added. Thus, the second datalanguage statement arranges for the data to be accepted and stored in that set of data items, thereby adding it to the file.

The following statement deletes a set of observations from an open file:

```
DEL OBS WITH TEMP > 200
```

DEL, like ADD, is an operator that may appear in any expression.

Consider a program transmitting a series of updates to existing observations in the WEATHER file. Each update contains a (time, location) pair that uniquely identifies it, and then a new value for temperature or humidity, or both. The first part of the update is fixed format, and the remainder, which contains the new value for temperature or humidity, is variable format. In the variable part of the update, the data items are self-identifying. That is, if the item is a new value for temperature, it begins with a code that identifies it as such. A set of these updates then looks like:

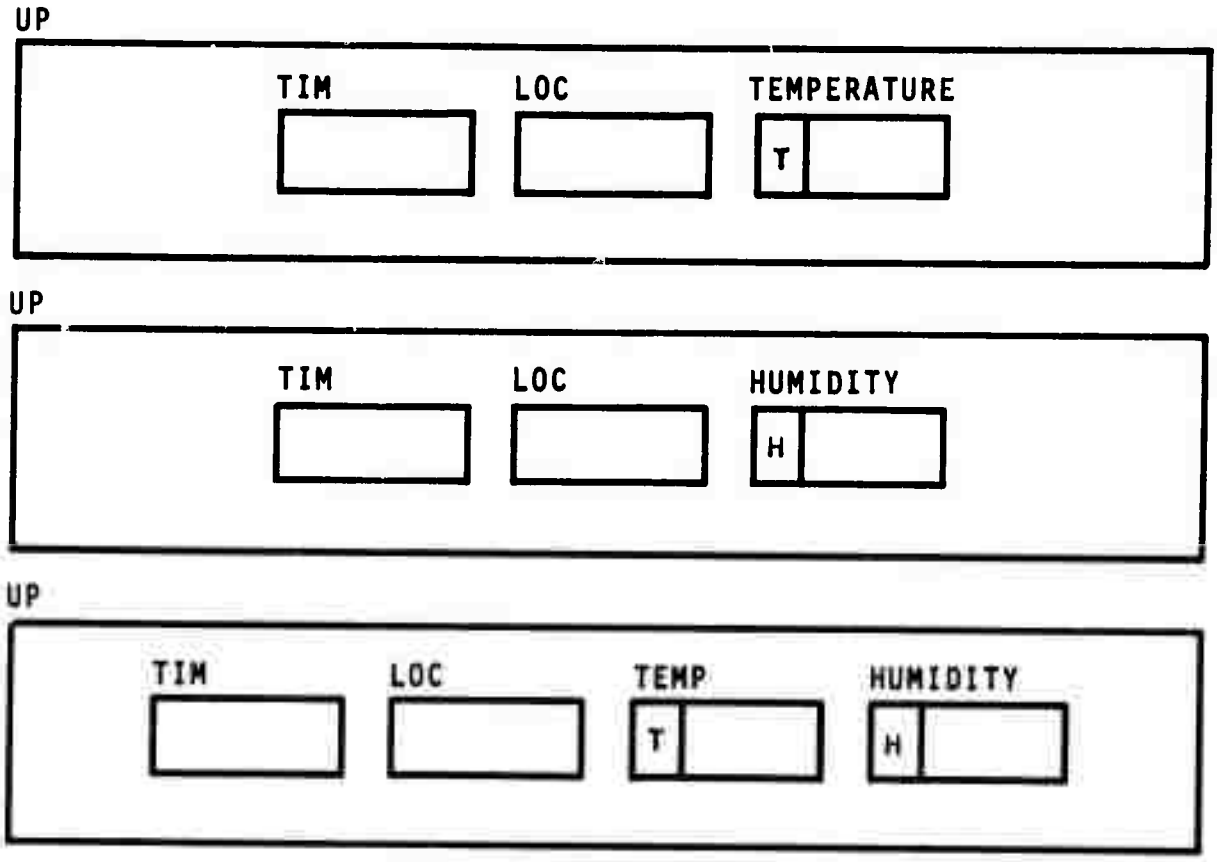


Figure 1.6

The information in figure 1.6 is encoded in a data description, and this is associated with the input port Y. Then the following datalanguage statements are sent to the datacomputer to perform the update:

```
OPEN CCA.RAW.CLIM.WEATHER...
(OBS WITH LOCATION = UP.LOC AND TIME = UP.TIM) ← UP
```

The second statement above performs the entire update. It inputs an UP, retrieves the corresponding OBS, updates it, and returns it to the file in place. There are several less concise ways to indicate this update in datalanguage. The one above is the most concise and leaves the most up to the datacomputer in terms of decision-making and optimization. Input through the port is done automatically as part of the process of looping through the set of UPs. An example of a less concise way to indicate the same operation is:

```
    OPEN CCA.RAW.CLIM.WEATHER,...
    BEGIN
    DEFINE S,OBS
    FOR UP
A:  GET S,R
    IF R.LOCATION = UP.LOC AND R.TIME = UP.TIM
    THEN R ← UP
    ELSE GO TO A
    END
    END
```

This set of statements accomplishes the same result as the previous example, but it tells the datacomputer <u>how</u> to conduct the search and <u>how</u> to coordinate the input, search and update operations. The

strategy that the datacomputer would choose if left to its own devices will always work. When the user has access to the right information, he may be able to choose a more effective strategy than the datacomputer. The one programmed above is most effective when the WEATHER file must be sequentially searched to locate the OBS corresponding to a given UP. (This is not true if the appropriate body of secondary information, such as an inversion, exists, and the file is large). To completely analyze the above example requires understanding much of the rest of this paper. To highlight:

1. A BEGIN-END block is used to force the interpreter to treat the set of statements as one statement, and to limit the scope of the definition of the set, S.

2. S is the set of all observations. This set is defined and named so that the GET operator (#5, below) can be used.

3. The FOR loop iterates once for each UP that is input. UP has been previously associated with port Y, so it's clear where to get the UPs from.

4. A: is a statement label.

5. GET gets the next thing from the set S, and associates the reference name R with it. That is, R stands for the current OBS from the set S. R.LOCATION means the location in the current OBS. UP.LOC means the location in the current UP.

6. Assigning UP to R does the update operation. It is a case of aggregate assignment. Elements of the aggregates UP and R with matching names are assigned to one another.

A data stream is the sequence of data items passing through a datacomputer port. In the examples so far, two techniques have been used for handling the data streams. In the first two examples, no description was associated with the stream prior to operating on it. Fully described data elements were simply stuffed through the port. In the last example, the data stream was described ahead of time, and the things in the data stream (UPs in this case) were then assigned to other things. The act of successively accessing the UPs caused the i/o operation to occur.

So far, we have concentrated on data manipulation at the expense of data description in the examples. Let us now briefly discuss data description. Refer back to figure 1.5, which shows WEATHER. It consists of many identical items called OBS. OBSs can be added to and deleted from WEATHER. In datalanguage, WEATHER is called a list.

An OBS has quite a different structure than a WEATHER. All OBSs have the same four named components (location, time, temp, and humidity) which are not identical to one another. An OBS is called a structure.

WEATHER is a list of structures. Each structure on the list contains four elementary data items of which three are numbers and one is a string. The data description for WEATHER is:

```
WEATHER,LIST
OBS,STRUCT
LOCATION,STRING
TIME,INT
TEMP,INT
HUMIDITY,INT
END OBS
```

An UP is similar in structure to an OBS, except that the last two elements are optional and self-identifying. This is indicated with a parameter following the datatype:

```
UP,STRUCT
LOC,STRING
TIM,INT
TEMP,INT,ID ← 'T'
HUMIDITY,INT,ID ← 'H'
END UP
```

The data descriptions above are not complete. In a real data description, many of the lines would have additional parameters, some of which are discussed in the section on data description. However, the crucial elements of the data description are name and datatype, and these are shown.

Data descriptions like the ones above are originally introduced to the datacomputer in declarations. Once available, they may be used in a variety of ways:

1. A data description can be associated with a file in a CREATE statement. The CREATE statement enters the file in the file directory, causes space to be allocated, and stores a data description in a special part of the file. An example of this process is:

```
DECL
WEATHER,LIST...
OBS,STRUCT...
LOCATION,STRING...
TIME,INT...
```

```
TEMP,INT...
HUMIDITY,INT...
END OBS
END DECL
CREATE CCA.RAW.CLIM.WEATHER,DESCR ← WEATHER....
```

The datalanguage between the DECL and the END DECL is a data description of something named WEATHER. The CREATE statement references this description, creating the file and saying that the data the file will contain will conform to this description. In the future, opening this file makes the description available.

2. A data description can be associated with a logical port when the port is established, using the PORT statement:

```
PORT X,external name, DESCR ← OBS...
```

The port statement above establishes a port named X, and specifies that data fitting the description of OBS will be transmitted through the port. The external name is something that associates X with some real data source in the outside world.

Note

The datacomputer will have a capability to accept datalanguage in any character set sufficient for its expression. The datalanguage port has a data description that states character set, how statements are terminated, etc. Precisely how this description is accessed by the user and altered is not clear yet. It will probably be done with a combination of protocol and datalanguage mechanisms.

3. A data description can be associated with a temporary that is available for holding intermediate results during data

management operations. Such temporaries can be manipulated in datalanguage in the same way other data is. Temporaries are established with the ALLOC statement:

```
ALLOC T,DESCR ← OBS,...
```

The temporary named T is created. It fits the description of an OBS. T can be assigned values and operated upon. It is not entered in the file system, and is not saved.

Some important parameters are associated with data description, and these are covered in a later section of this paper. They include the specification of the character set for strings, the length of the item, the format of a floating point number, the formatting of the data stream, and options for less visible structures (such as the inversion, for rapid retrieval, or the overflow areas, for sorted files). In short, these parameters specify everything needed to understand and operate on the data in datalanguage.

This completes a brief review of the data manipulation and description facilities in datalanguage. The emphasis in the remainder of the paper is on the concepts of the design, on the rules that determine what can and cannot be written in datalanguage, and on the properties of the various datalanguage facilities.

Chapter 2
Data Description Concepts

2.1 Data Containers

In the introduction, data structures are illustrated as groups of nested boxes. The data itself is not pictured, but the illustrations imply that the data is contained "inside" the boxes.

In the remainder of this paper, as much interest will be focused on these boxes as on the information inside them, and it will be necessary to distinguish between the boxes and the data itself. For this reason, we introduce the term data container which is defined:

1. at the conceptual level, a data container is simply an imaginary box in which either data or other data containers are kept.

2. at the physical level, a data container is a named set of locations in some storage medium. Data containers exist in core, on direct access storage devices, on tape, etc.

3. in a logical, but somewhat more concrete sense, a data container is a file, a logical record, a field, a variable in a Fortran program, an array in a Fortran program, etc.

From definition 3, above, data container is simply a general term for a familiar class of things. The familiar class, unfortunately, has no familiar name that fits just right in datalanguage. Such a class name is needed, however, in discussing datalanguage, because datalanguage has so many facilities defined on the entire class. For example, the datalanguage facilities for adding things to a

list or deleting them from a list can be applied to files, to records, to fields, to in-core lists -- in short, to any data container that has the structure of a list.

2.2 Types of Data Containers

There are 8 types of data containers: STRING, INT, REAL, PTR, ARRAY, LIST, STRUCT, and MIX.

Simple Containers

Simple containers (types STRING, REAL, INT and PTR) hold single data items.

STRING

A STRING container holds a single string. A string is a series of 0 or more characters from the same character set. Two character sets are predefined in datalanguage: BIT and ASCII.

BIT contains two 1-bit characters: '0' and '1'.

ASCII contains 8-bit characters, including upper and lower case alphabetics, numerics, and an assortment of special characters.

Additional character sets can be defined in datalanguage. These have characters of any length up to some implementation-defined maximum number of bits. When a character set is defined, sufficient information to make translations between it and a predefined character set is provided in the definition.

INT

An INT container holds a single whole number, expressed as a binary integer in a standard way. This standard is twos-complement notation. Integers may be any length up to some implementation-defined maximum (perhaps 256 bits).

REAL
A REAL container holds a single floating point number, expressed in one of a set of standard formats.

PTR
A PTR container holds one "pointer" to another container. The pointer is a name or address for the other container. Currently, PTR is not fully defined.

When a particular container is defined, it is given a name by which it can be referenced in datalanguage, its type is specified, and a large number of other properties are determined. Consider a container created to hold the names of cities, which can be defined:

```
CITY,STRING(ASCII),LMAX ← 25
```

In this definition, the first thing is the name, the next is the container type, and the last is the specification of a certain parameter (LMAX). The name is CITY. The type is STRING(ASCII), or string of ASCII characters. The parameter, LMAX, is the maximum length of the string to be held in the container. In the definition above, LMAX is set to 25 (characters).

Aggregate Containers
Next consider creating a container to hold the capitol city of each state in the US. Fifty identical containers would be desired, one for each state. Situations like this motivate the inclusion in datalanguage of container types ARRAY, LIST, STRUCT, and MIX.

ARRAY
An ARRAY container holds some fixed number of other containers in some fixed order. All the inner containers, or elements of the array, have the same name and type, and are therefore quite similar.

Elements of an array are never added, deleted or reordered. However, the values of elements can be changed.

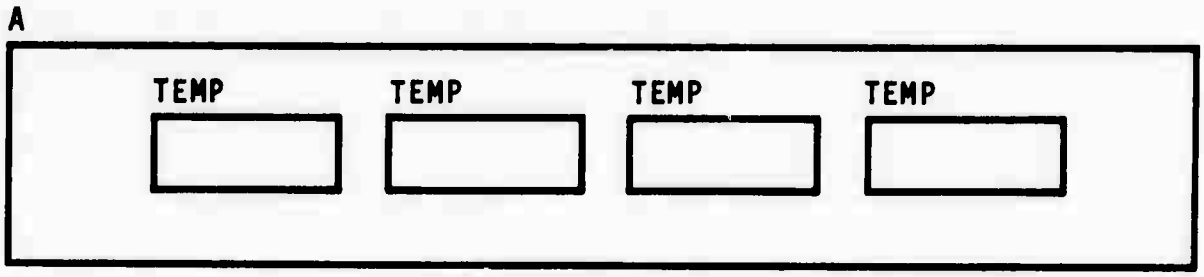


Figure 2.1

The array in figure 2.1 holds 4 containers named TEMP.

LIST

A LIST container holds a number of other containers of a single name and type. As in the array, the elements of a list are quite similar to one another. However, the number and order of the elements of a list may vary over time as elements are added and deleted.

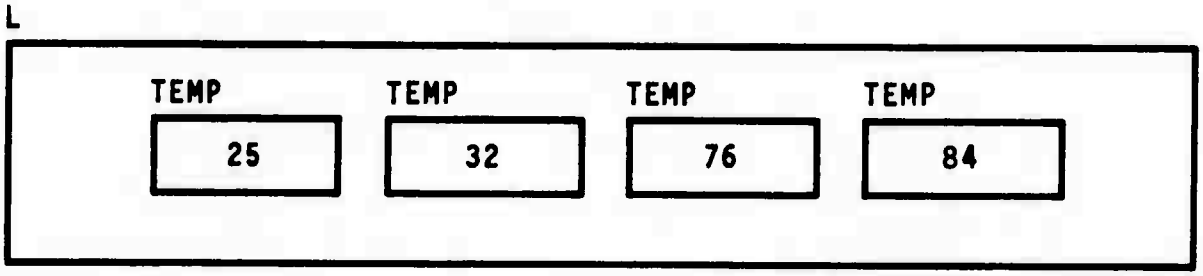


Figure 2.2

The list, L, above holds four containers named TEMP. Below is the same list after adding two TEMPs and deleting one other.

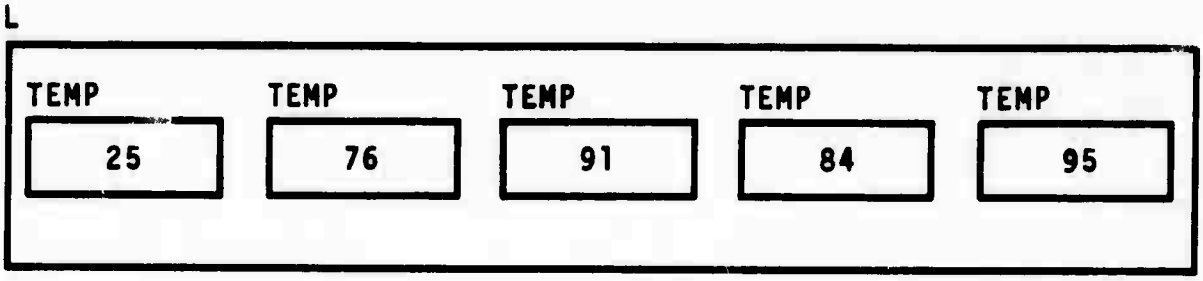


Figure 2.3

STRUCT

A structure, or container of type STRUCT, holds a number of containers that have unique names and may have dissimilar types. The elements of a structure always occur in the same order, and never occur more than once in the structure.

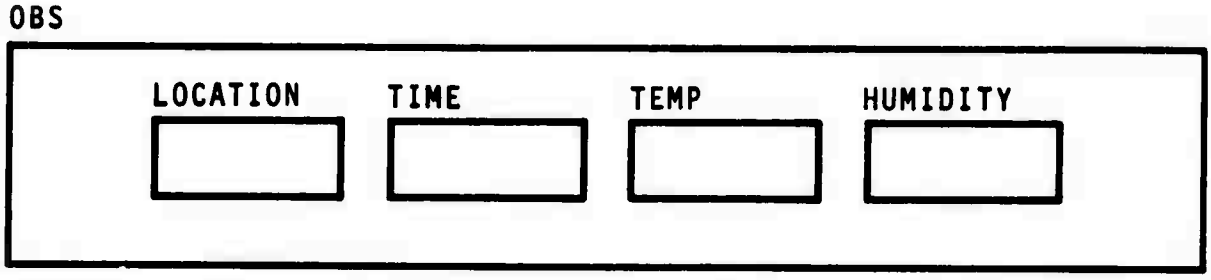


Figure 2.4

MIX

A mixture, or container of type MIX, holds a number of containers that may be dissimilar from one another, and are self-identifying. The elements of the mixture may appear in any order and may occur any number of times. New elements may be added and elements may be deleted.

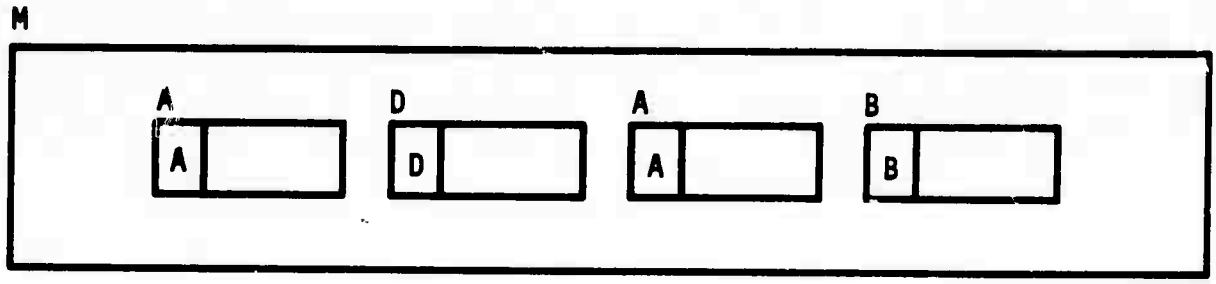


Figure 2.5

The mixture, M, holds four containers. Present are two As, a B and a D. Though it can contain a C, none is present in this M.

With suitable nesting of containers, most of the file structures that occur in data management systems can be created using data-language arrays, lists, structures and mixtures. Fields are usually one of the simple container types (STRING, INT, REAL, PTR). Records are usually structures (STRUCT) or mixtures (MIX) of fields. Files are usually lists of records.

2.3 Container Names

Figure 1.5 shows a large number of containers named TEMP. Each TEMP is part of an OBS which is part of WEATHER. These TEMPs can be referenced by any of the following names:

```
TEMP
OBS.TEMP
WEATHER.TEMP
WEATHER.OBS.TEMP
```

The last name is the full name of TEMP, and is used internally by the datacomputer. In the language, however, any of the components of the full name can be omitted for convenience. They are used only to resolve ambiguities.

Chapter 3
Data Storage and Retrieval Concepts

3.1 Functions

Most data storage and retrieval is done in datalanguage with functions.

The inputs and outputs of functions are sets of data containers. A set is simply a collection of zero, one or many data containers.

Thus a function is something that inputs a bunch of data containers and outputs a bunch of data containers a little later. (Note that this is suspiciously similar to what a datacomputer does.)

Since the inputs and outputs of all functions are sets of data containers, functions can be connected to one another rather indiscriminately. This is a great advantage in designing the language, as it permits considerable uniformity and flexibility in the notation.

Equal is a familiar example of a function. It is used in several of the examples in chapter 1 (recall TRANS X,OBS WITH LOCATION = 'BOSTON'). It is invoked whenever a datalanguage expression contains the equal sign, "=".

The datalanguage equal function acts like a filter. It inputs a set of data containers and outputs a smaller set. Its method of filtering is to compare the value of the data in each container to a filter value. If they are equal, the container passes through the filter and is output. If not, the container is not output. Figure 3.1 illustrates the operation of the equal function when the filter value is "transistors". At input port 0, the set of containers

to be filtered is presented. At input port 1, the set of filter values is presented. The containers having the value "transistors" emerge at the output port.

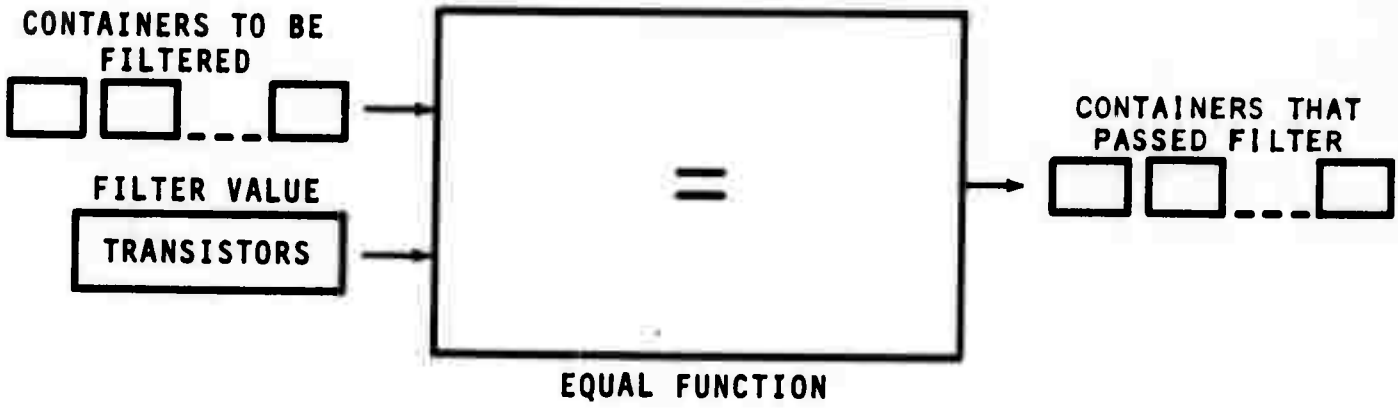


Figure 3.1

Functions are, of course, computer programs that run on the datacomputer. Some are built into the system and written by the implementers of the datacomputer. These are built-in functions. Some are written by users in datalanguage, and stored in the datacomputer or input when they are used. These are user functions.

The form of illustration in figure 3.1 is particularly convenient for presenting certain aspects of functions in this paper. It is not meant to reflect precisely data movement inside the datacomputer. The operation in figure 3.1 can be performed by the equal function, in some cases, without access to the data containers at all. An

example is when a secondary body of information (about the values of the containers) exists. In other cases, the containers are individually accessed and checked. This accessing process may be carried out using a variety of buffering schemes, or even by several independent processes operating in parallel. However, in all such cases, the result is the same. It is this result that is emphasized in the diagram, and in that part of datalanguage concerned with functions.

3.2 Operators

The operators are an important class of datalanguage functions. They differ from other functions in that a special notation is used to invoke them in the language. This notation is one of:

prefix form	operator operand0
infix form	operand0 operator operand1
implied form	operand0 operand1

enclosure form, which is explained later in this section.

An example of an infix form operator is equal, written:

```
TEMP = 75.
```

Here, "TEMP" is operand0, "75" is operand1, and "=" is the operator.

A prefix form operator is arithmetic negation, written:

```
-TEMP
```

Here, "TEMP" is operand0 and "-" is the operator.

An example of implied form is:

A(25)

Here "A" is operand0, "(25)" is operand1, and the operator, subscript, is not written. Implied form is permitted only for function call and subscript.

A. Comparison operators

The comparison operators are equal, less than, greater than, not equal, not less than, and not greater than. They all function as filters, in the fashion described in section 3.1.

The following is an invocation of the equal operator:

TEMP = 75

Here, operand0 is TEMP, the set of all containers named TEMP. Operand1 is 75, a set of one container with value 75. In figure 3.2, the equal operation is diagrammed.

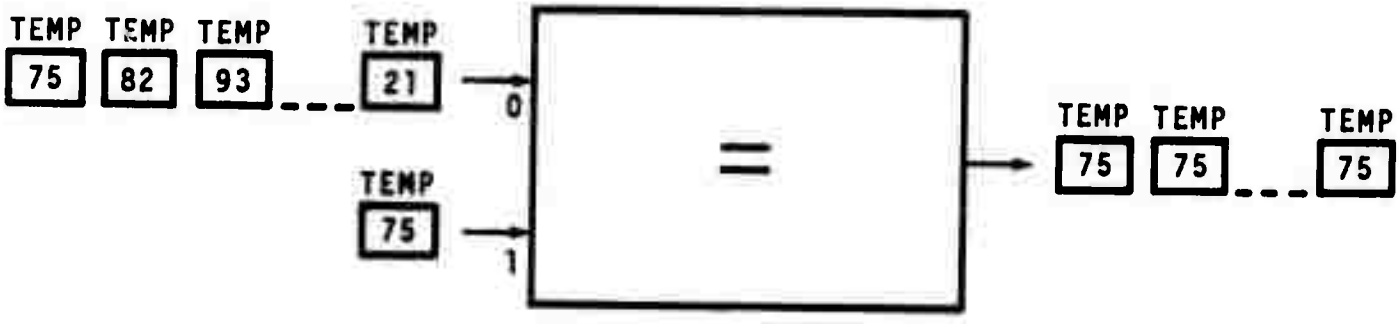


Figure 3.2

It is important to understand that the expression TEMP = 75 is a reference to a set of containers in exactly the same sense as TEMP is a reference to a set of containers. This is because TEMP = 75 represents the result of calling a function, and the output of all datalanguage functions is one or more data containers.

In the example above, we show a single data container passing through port 1 of the equal function. The value of this data container is used during the entire filtering process. This is not the only fashion in which comparison functions can operate. It is also permissible to input a set of more than one data containers through port 1. In this case a correspondence (between the containers entering at port 1 and the containers entering at port 0) is defined. A container through port 1 is input, and its value is used to "load the filter". Then the corresponding containers are entered at port 0, and the ones that fit through the filter exit at the output port. The cycle is repeated until one of the input sets is exhausted. For an example, return to the WEATHER container that we have been using. Each OBS contains a TEMP and a HUMIDITY. Assume for a moment that HUMIDITY is expressed as an integer, ranging from 0-100, and TEMP is usually in this range also. If someone is interested in finding all the cases in which TEMP is greater than HUMIDITY, he might write:

```
TEMP > HUMIDITY
```

which is legal datalanguage. This invokes the greater than function, which outputs a set of TEMP containers (figure 3.3).

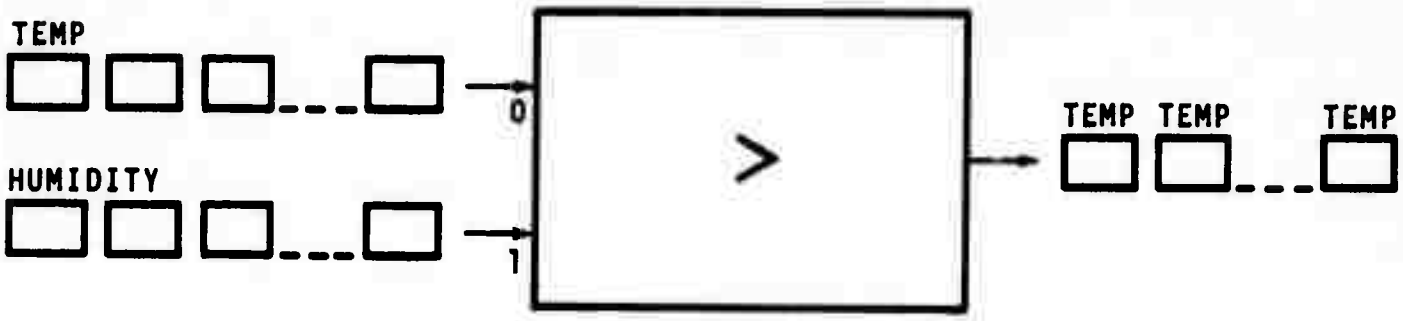


Figure 3.3

In this example, the correspondence between the containers in the TEMP set and the containers in the HUMIDITY set is obvious: there is one and only one of each in each OBS, and the ones from the same observation correspond.

In this latest example, it is TEMPs rather than HUMIDITYs that the function puts out. This is because the function is defined to put out the containers that come in through port 0 and filter with the containers that come in through port 1. It is a rule of the language that the operand0 (in an expression like TEMP>HUMIDITY) is associated with port 0, operand1 with port 1, etc. Someone wanting the HUMIDITYs in the above case could write HUMIDITY<TEMP.

B. Assign

The assign operator puts values into containers. When a container is assigned a value, previous values are lost. Consider the assignment of the value 'NYC' to all containers named LOCATION:

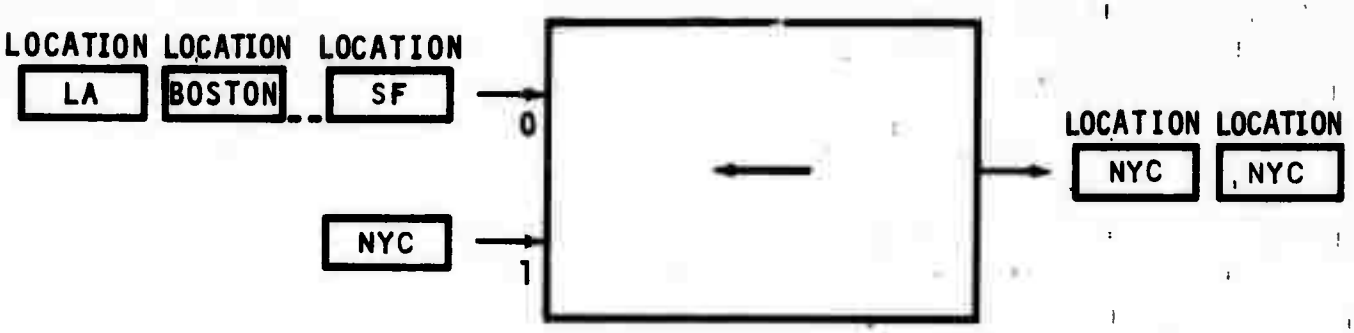


Figure 3.4

A set of one container with value 'NYC' is input through port 1. The set of all containers named LOCATION is input through port 0. All the containers that enter at port 0 exit at the output port, with their value set to 'NYC'.

An arrow pointing to the left is used to indicate assignment in datalanguage. The example above is written:

```
LOCATION ← 'NYC'
```

Either operand to the assignment operator can be a set of one or many containers.

Notice that the assignment operator <u>does produce output</u>. LOCATION ← 'NYC' is a reference to a set of containers, just as LOCATION is. Thus, the following is syntactically correct, and even sensible: A←(B← 'NYC').

Figure 3.5 should make this case clear:

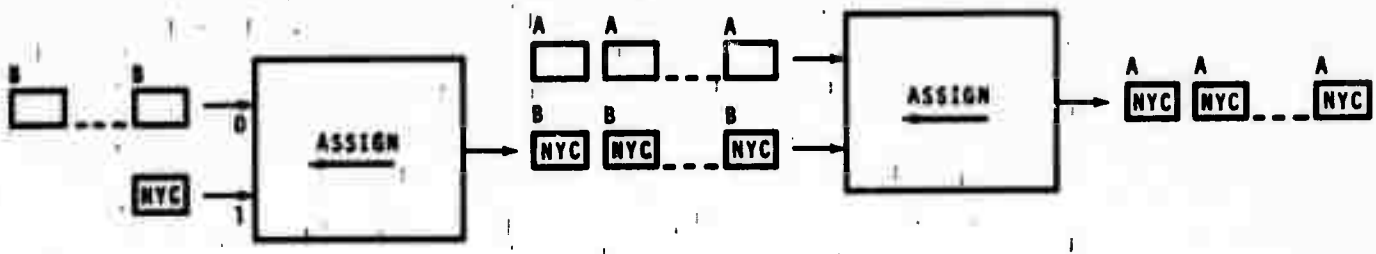


Figure 3.5

First the B's are assigned values of 'NYC' and then the A's corresponding to them are assigned the values of those B's. So the expression A← (B← 'NYC') might be read "assign the B's the value 'NYC' and then assign the A's the values that the B's have".

Another illustration of the coupling of two functions is:

(TEMP = 75) ← 92

This expression changes to 92 the value of all TEMP's having the value 75. Figure 3.6 should make this clear:

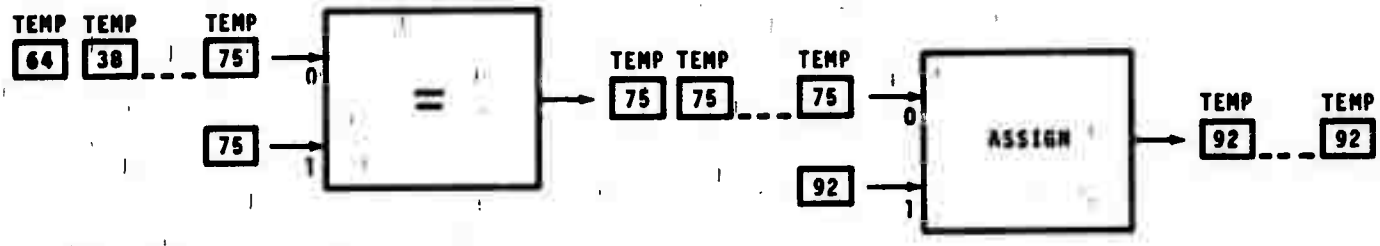


Figure 3.6

All the TEMP's are input to the equal function, and a subset of them emerges. This subset is then input to the assign box, and emerges with new values.

C. Relational operators

The relational operators, like the comparison operators, are filters. However, they have different criteria for passing or filtering containers. They pass or filter depending on whether the container in question contains, does not contain, is contained by or is not contained by some other container.

Applications of the relational operators are abundant. Using the comparison operator, a reference to the set of TEMPs with value greater than 72 can be generated:

TEMP > 72

A relational operator is then used to reference the set of OBSs that contain those TEMPs:

OBS WITH TEMP > 72

The line of datalanguage above generates a reference to the OBSs that contain some TEMP greater than 72. Figure 3.7 diagrams the process.

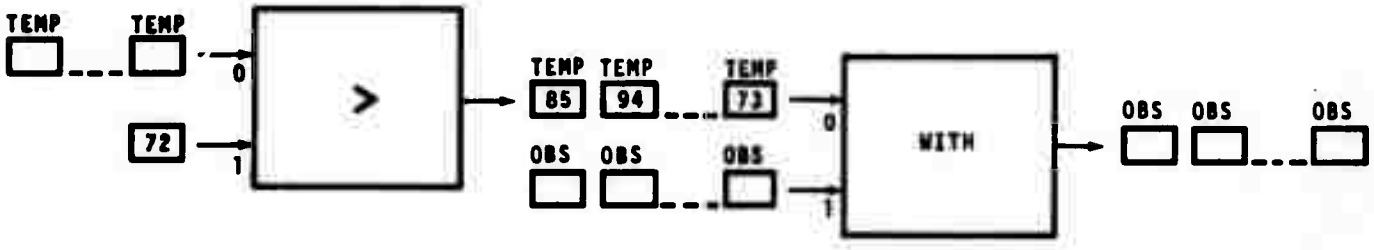


Figure 3.7

The process starts at the left of the picture, with all the containers named TEMP passing in through port 0 of the greater than function. Only those with values greater than 72 emerge at the output port. These are input through port 1 of the WITH function. At port 0, all the OBSs are input. The OBSs that emerge through the output port are the ones that contain a TEMP in the input set.

The opposite filtering effect can be produced by writing:

OBS WITHOUT TEMP > 72

This expression references all the OBSERVATIONS that do not contain a temperature greater than 72.

Thus, in the examples above, WITH means that contain some, and WITHOUT means that do not contain some. A separate pair of operators could be defined for that are contained by some and that are not contained by some. However, the potential relationships between any pair of containers (like an OBS and a TEMP) are clear from the definitions of those containers. Thus WITH and WITHOUT can be used to write either contain or contained by, and are always unambiguous.

D. Boolean operators

The datalanguage boolean operators are AND, OR and NOT.

AND and OR have two input ports and one output port. A set of containers comes in each of the input ports. AND outputs the intersection of the two sets. OR outputs the union. That is, AND outputs the containers that occur in both sets, while OR outputs the containers that occur in either of the sets or in both sets.

NOT inputs a single set and outputs the containers that are not in that set but are in the universe of reference. Universe of reference will be defined later.

An example of an expression involving AND:

```
(OBS WITH LOCATION = 'NYC') AND (OBS WITH TEMP = 72)
```

This expression can be written more naturally by distributing the relational operator:

```
OBS WITH (LOCATION = 'NYC' AND TEMP = 72)
```

The datalanguage compiler can recognize the equivalence of the two and compile the latter as the former.

E. Computational operators

We use computation somewhat loosely to mean string manipulation as well as arithmetic computation. Computational functions do not output the same containers that they input. They generate temporary containers whose values bear the specified relationship to the values of the input containers. The generated containers inherit a great many properties from the input containers, including the relational properties of the inputs from port 0. The computational operators are +, -, *, / and || (concatenation). With this class of operators there are a thoroughly researched set of standard solutions to most of the problems that arise, and they warrant no discussion here. Expanding this class to include other primitives, particularly for string handling is an obvious possibility that will be investigated.

The only noteworthy property of datalanguage computational functions is that they are defined for sets the same way comparison functions are.

F. Subscripting

Subscripting is selection by element number. Consider the container WEATHER, in figure 1.5. It is a list of OBSs, which we can number successively from zero. Using subscripting, the first can be referenced as WEATHER(0), the second as WEATHER(1), etc. The same technique can be applied within the OBSs. The first element of each OBS can be referred to by its name (LOCATION), or by its element number, which is zero. Thus OBS(0) is equivalent to LOCATION and the two can be used interchangeably.

The technique of subscripting can be applied to any type of data container, and can extract either single data containers or groups of them. It can also be applied to sets of data containers, to select, for example, the first, third and fourth containers from a set. The different uses of subscripting are examined in order of increasing complexity, below.

First consider the case of container subscripting, which is the more common one. The simplest application is the selection of a single element from a single container. The container from which the element is to be extracted is presented at port 0 of the subscripting function. The number of the element (normally numbering from 0, but more about this later) is presented at port 1. The desired element then emerges from the output port:

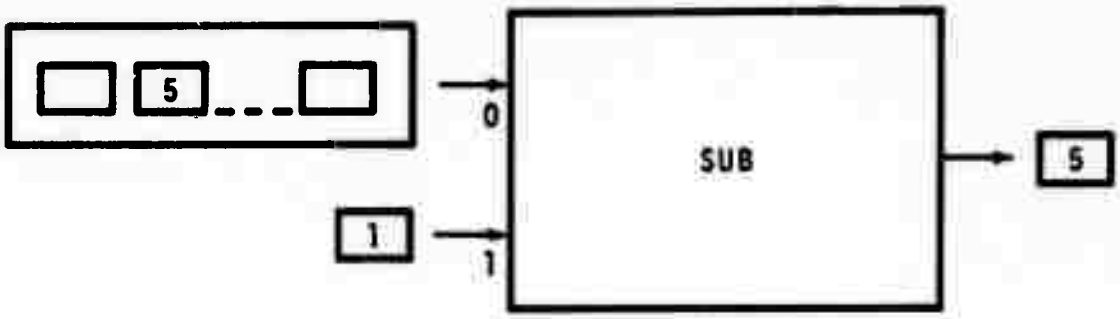


Figure 3.8

Since functions always operate on sets of containers, this first example is merely the degenerate case in which both input sets are composed of one container each. As with all the other functions, the input sets may contain more than one container, so long as an appropriate correspondence can be defined.

The next case is the selection of more than 1 element from a container. This produces a new container that has only the elements selected. The new container is a generated container that inherits most of its properties from the original, including container type. The selection of more than one element is accomplished by presentation of a list of numbers at port 1. Each of the numbers is applied to the container input at port 0, and each selects a component:

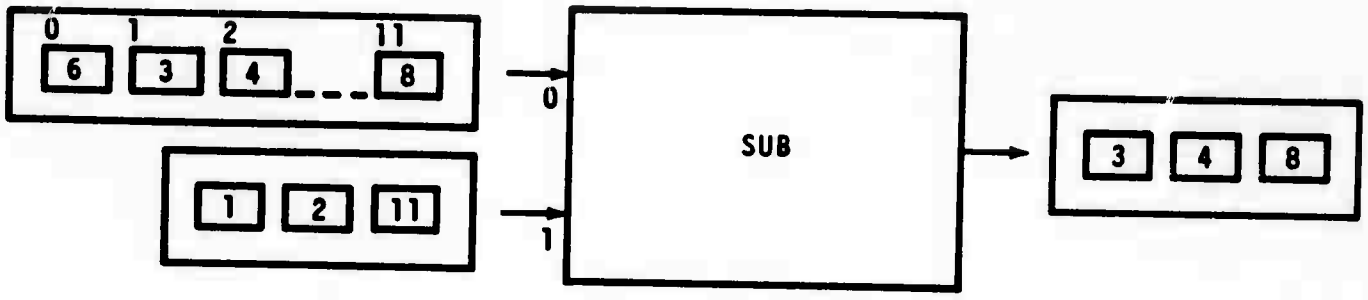


Figure 3.9

When more than one container is selected from an aggregate, the result is an aggregate container. It is important to realize that this result has the same datatype as the input -- it is a structure if the input is a structure, etc. This is distinct from the first case discussed, figure 3.8, in which the output is a simple container, which existed before the subscripting operation and has an established datatype.

The list that is input at port 1 (i.e., the one that supplies the subscripts) can also be a virtual list. The virtual list is a special datalanguage construction that acts exactly like a list of integers, but takes up very little space.

It is a specification for a simple iterative function that generates the list, similar to the way the Fortran DO generates values for a variable to drive a loop. The list generation function involves a variable, which is named if it is to be referenced. The input to the function is a group of expressions. One gives an initial value to the variable, one gives a value to be used to test for end of list, and a third gives a value that is to be added to the variable after

each list entry is generated. These expressions are called initial, endtest, and increment, respectively. Two other expressions, while and until, cause end of list if they either fail to or begin to return the null set, respectively.

A virtual list is written:

(initial : endtest : increment : variable : while : until)

A few examples:

(1:10) generates the list 1,2,...10 (increment defaults to 1)
(1:10:3) generates the list 1,4,7,10
(1:10:I:I) generates the list 1,2,4,8

In container subscripting, virtual lists are handy in forming sub-arrays. For example, if A is an array of 25 strings, then the following expression forms a smaller array containing only the last 5 strings:

A(20:24)

Every fifth string is obtained with:

A(0:24:5).

Set subscripting (operator SUBSET) is always written explicitly. Its input at port 0 is a set of containers. Its input at port 1 is a list or virtual list of numbers. It leaves all the input containers intact, but acts as a filter on the set, outputting only those containers whose position in the set corresponds to a number on the list.

Thus OBS SUBSET (0:9) is a reference to the first 10 OBSs.

G. Function call

Function call is an operator that manages the passing of arguments to functions. It takes a function name as input at port 0, and a data container or set of data containers at port 1. Each container input at port 1 is one complete set of arguments for the function to be called. The function call matches the types of these arguments to the types required by the function, brings about a conversion if necessary, and then presents them at the appropriate ports of the function. The output port of the function returns containers to the function call, which passes them through its output port:

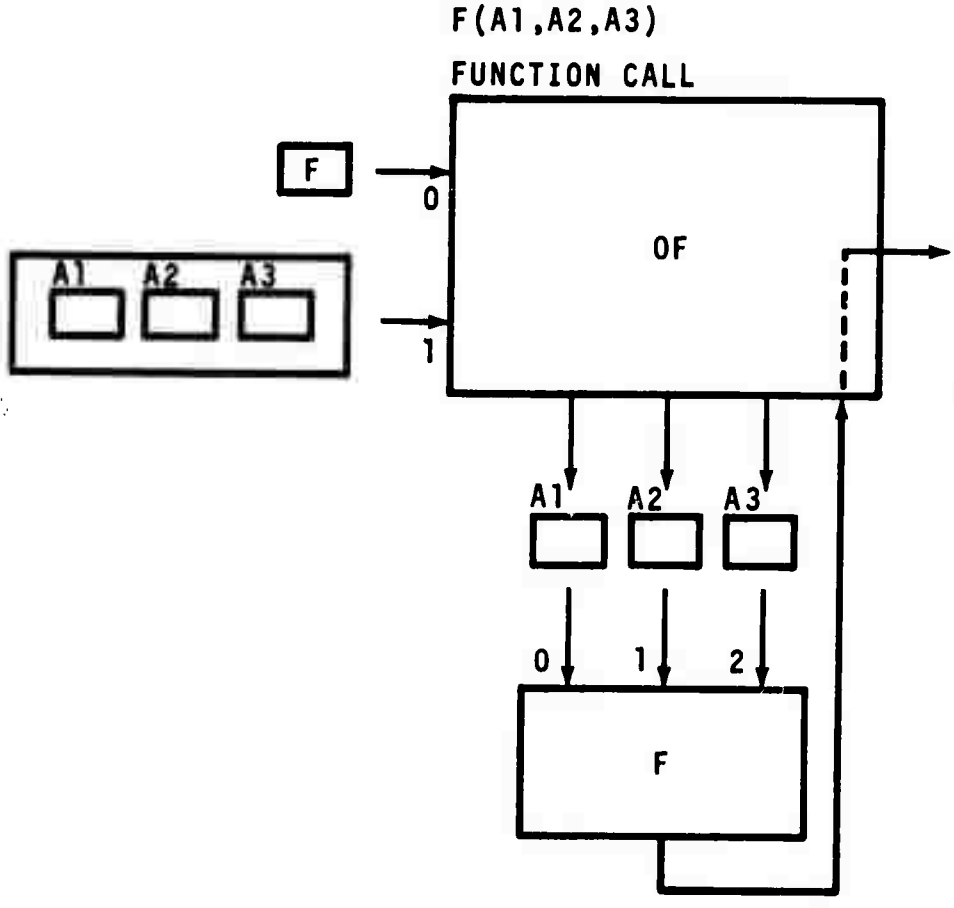


Figure 3.10

H. Enclosure

Enclosure is an operator with an arbitrary number of input ports and one output port. In a cycle, it takes one container from each input port, encloses them together in a new container and puts that out through the output port. The generated container is always a structure (STRUCT). The input ports accept corresponding sets of containers. When there is more than one container in each of the input sets, then there is more than one container in the output set.

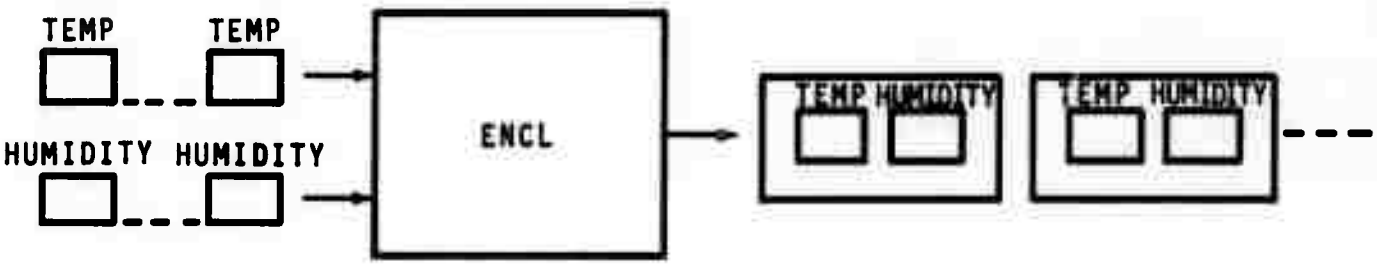


Figure 3.11

Enclosure is indicated by listing the sets of containers to be enclosed, separating the items of the list by commas and delimiting it with parenthesis:

(TEMP,HUMIDITY)

Creates a set of containers that each contain one TEMP and one HUMIDITY.

I. List/Mixture maintenance

Entries can be added to and deleted from lists and mixtures. Thus there are two operators that are defined only for sets of containers that are members of lists and mixtures. These are DEL and ADD.

DEL removes a set of containers from a list or mixture:

```
DEL OBS WITH TEMP > 90
```

ADD creates a set of containers, adds them to a list or mixture and returns a reference to them. Thus:

```
ADD OBS
```

is a reference to a set of observations that have never existed before. This set can be assigned values in the usual way, as in:

```
ADD W1.OBS ← W2.OBS WITH TEMP = 50
```

This statement adds a set of observations to list W1. The new observations will be identical to the observations on list W2 that have temperatures of 50.

The ADD operator has a property that sets it apart from the others discussed so far. The size of a set of containers generated by ADD is determined by the context in which ADD is used, not by the operand of ADD. In the example above, ADD W1.OBS generates a set of W1.OBS. The number of observations in the set, however, is determined by the size of the set that is assigned to ADD W1.OBS. If the context does not supply a size for the indeterminate set, it is given a size of 1. Thus in most contexts, the expression ADD OBS will create a single new container.

In some applications it is desirable to impose an order on a list or mixture. When this order is the same as that in which entries are added, no extra effort is required. (The order of items in lists and mixtures will be maintained unless explicitly disturbed.) However, there may be some order imposed, as when a list is maintained in alphabetical order by one field, while the entries of the list are added in random order. Or, the list may be ordered by some property that is not apparent from the content of the entries, or at least is not trivially derived from them (i.e., trivial in "seconds of cpu time"). When the ordering of the list is based on a simple function of the contents of the entries, and that function is specified in the data description of the list, then the normal ADD function of datalanguage can be expected to cause a new entry to be added in the right place. In all other cases, the point of addition must be indicated by one of the special purpose ADD functions, ADDA and ADDB. These are not operators (that is, they must be written in function call notation). They take a pair of arguments. One is the kind of container to be added to the list or mixture, as in ADD. The other is a pointer to a particular object on the list. ADDA adds after the container pointed to. ADDB points before.

3.3 Built-in Functions

A large library of built-in functions - for explicit control of data conversion, for accessing data descriptions, for obtaining status information and for a host of other activities - will be present in the datacomputer. The information returned by these functions can be input to other functions, used in conditional statements to determine action, or simply transmitted to the remote program.

3.4 Named Sets and Set Formation

Evaluating the functions discussed in section 3.2 involves the use of most of the data management techniques employed in the datacomputer. Storage, retrieval, file searching, and all manners of optimization techniques are used. Since all the functions operate on sets of containers, one of the most crucial parts of the evaluation process is the efficient formation of these sets.

Set formation is the process of determining the membership of a set and the path of access to each member. Latalanguage has a facility for defining and naming sets, and then controlling the method in which they are formed. Once formed, explicit control can be excercised over their use (see last example in section 1.3).

The facilities for controlling set formation are discussed briefly in Chapters 4 and 5. They will be defined more fully in the early stages of implementation.

Chapter 4
Elements of the Language

4.1 Statements

The statement is the unit of expression in datalanguage. When a complete datalanguage statement is received by the interpreter, it is evaluated. The interpreter is then ready to accept another statement. There are six types of datalanguage statements:

1. the expression statement requests the evaluation of a datalanguage expression. An expression statement is simply an expression optionally preceded with one or more statement labels. (Expression is defined in 4.2.)

2. the keyword statement requests the evaluation of a keyword function. The syntax and semantics of the keyword statement are determined by the keyword that begins it. An example is GOTO, which effects a transfer of control.

3. the declaration statement presents one or more data descriptions to the system. These descriptions are analyzed and remembered, and then may be referenced.

4. the if statement requests the conditional evaluation of a statement.

5. the for statement requests the iterative evaluation of one or more statements contained therein.

6. the begin statement requests the evaluation of the statements contained therein, and the treatment of these statements (externally) as though they were a single statement.

4.2 Expressions

An expression is a reference to a set of data containers.

The name of a data container is an expression.

```
OBS
```

is a reference to the set of containers named OBS.

Any syntactic combination of operators and operands is an expression:

```
TEMP ← 72
OUTPUT ← OBSERVATION WITH LOCATION = 'CAMBRIDGE'
HUMIDITY < 25
```

The first two of the above expressions accomplish something aside from their reference to a set of containers. The last is merely a reference. All are perfectly good expressions.

The datalanguage operators have precedences used to govern the parsing of expressions. The default association of operators and operands (through precedence) may be overridden through the use of parenthesis.

Constants are written in expressions in a format that depends on their datatype:

```
123 is an integer in the prevailing radix
'ABODE' is a string in the prevailing character code
1.567E4 is a real number in the prevailing real type
```

X'123' is an integer in the radix defined to have mnemonic X

E'PQRSTU' is a string in the character set defined to have mnemonic E

I'1.3E-2' is a real number in the real type defined to have mnemonic I

Defaults and pre-defined mnemonics will be settled at a later date. A proposal for them is:

defaults - code ASCII, integer radix 10, real type PDP10
pre-defined mnemonics - B=bit, for strings
X=hexadecimal, O=octal, for radices

There will be one real type for each format on each machine. Using CODE and RADIX keyword statements, users define new mnemonics, redefine existing ones, and change the prevailing character code and radix.

The length of strings is self-evident. There will be a standard length for the numbers. When a non-standard length number is desired, it can be declared and named in a data description.

When a constant is used in an expression, it causes a set having 1 container to be generated. This container has the constant as its value.

4.3 Iteration

Iteration is accomplished by writing the statements to be iteratively evaluated between a FOR and an END. The FOR causes the statements to be evaluated once for each container in the set of containers referenced by an expression:

```
FOR OBS WITH LOCATION = 'NYC'
LOCATION ← 'NY, NY'
END
```

The statement LOCATION ← 'NY, NY' is evaluated once for each container in the set of containers referenced by the expression OBS WITH LOCATION = 'NYC'. Each time the statement inside the loop is evaluated, it operates on a single OBS from the set that is driving the loop.

The statement LOCATION ← 'NY, NY' has a slightly different interpretation inside the FOR loop than it does outside. Inside it operates on a single LOCATION container. If it appeared outside of a FOR loop, it would operate on all LOCATION containers.

In fact, the definition of the statement does not change. It still operates on all the LOCATION containers, but in a smaller universe of reference. The universe of reference is a function of the expression in the FOR statement. This expression will always reference some set of containers. The universe of reference inside the loop is made up of:

1. the current container in the set of containers driving the loop

2. all containers contained in 1.

3. all containers containing 1.

In the example above, we are looping on all the OBSs in a set of OBS. The universe of reference on a particular iteration is:

1. the current OBS

2. the LOCATION, TIME, TEMP, and HUM contained in that OBS

3. the list WEATHER that contains that OBS.

Within the loop, the set of all containers named OBS is a single container. So is the set of all containers named LOCATION.

A name will be interpreted with respect to the universe of reference immediately outside of the current loop, if it is prefixed with a "%". A pair of "%" pops two levels, etc.

```
1     FOR OBS WITH TEMP > 50
2     FOR %OBS WITH %LOCATION = LOCATION AND %HUM > HUM
3     X ← (LOCATION,HUM)
4     END
5     END
```

The lines in the example above are numbered for reference in this text. There are a pair of nested loops that result in the output, through port X, of (location, humidity) pairs. There is a set of such pairs for each observation with a temperature greater than 50. Each iteration of the outer loop shifts the universe of reference to a new observation in the set of observations that has TEMP > 50. The inner FOR then finds the set of observations that have the same location as and a greater humidity than the observation currently being referenced by the outer FOR.

In line 1, the universe of reference is the entire list of OBS. OBS means the set of all observations and TEMP means the set of all temperatures.

In line 2, the universe of reference has been reduced. LOCATION refers to "the set of all LOCATIONs in the current OBS", where

"the current OBS" is defined by the outer FOR statement. On the other hand, %OBS, %LOCATION and %HUM are interpreted with respect to the universe of reference they would have if they were not in a FOR loop at all. Thus %OBS means "all the OBS in the whole file". %LOCATION means all the LOCATIONs in the whole file, not all the LOCATIONs in the current OBS.

In line 3, the universe of reference has been changed again, because line 3 is contained in the inner FOR loop. Again, the loop is driven by a set of OBS, so LOCATION and HUM are interpreted with respect to the current OBS. However, the current OBS in the inner loop is not the same OBS as the current OBS in the outer loop.

4.4 Conditional Statements

The IF statement allows for conditional execution of a statement. Its form is:

IF expression
THEN statement
ELSE statement

The entire complex of keywords and statements is regarded as a single statement externally. Thus an IF statement can be contained in another IF statement, etc.

The expression following the IF is evaluated first. If it evaluates to a non-empty set, then the statement following the THEN is executed. Otherwise, the statement following the ELSE is executed. The ELSE and accompanying statement can be omitted.

4.5 Data Descriptions

A data description specifies the name and type of a data container, and optionally specifies a number of other parameters. An example is:

```
COLOR,STRING(ASCII),LMAX ← 20
```

The container named COLOR will hold a single ASCII string of up to 20 characters.

When a container holds other containers, these are described in parenthesis following the datatype:

```
WEATHER,LIST (OBS ... END OBS),....
```

or on the line(s) following:

```
WEATHER,LIST,.....
OBS,STRUCT,...
LOCATION,STRING,...
TIME,INT,...
...
END OBS.
```

Above, WEATHER is defined as a list of OBS. OBS is defined as a structure of (LOCATION, TIME,...). An END is used to mark the end of the description of OBS, since it is a structure and a structure can contain an arbitrary number of named elements, each of which must be described. There is no END for the list (WEATHER), since a list contains only one named element that repeats over and over. Since only one named element is needed to describe the contents of a list, no END WEATHER is required to parse the description. If

END WEATHER is present, it is ignored. Similarly, arrays need no END, while mixtures do.

Name and datatype are clearly the most important data description parameters, from the point of view of writing datalanguage requests. The other parameters rarely affect how requests are written or what they do conceptually. They do, however, affect the format of data transmitted in data streams, the structuring of files in a performance, rather than conceptual, sense.

The parameters that affect data formats are:

Length (L) the length of the object that can fit in the container. Length has a unit associated with the datatype:

integer	bits
string	characters
real	bits
ptr	bits
array	array elements
list	list elements
struct	bits
mix	bits

Maximum length and minimum length (LMAX and LMIN) define a range for L.

Delimiter (D) defines a thing that marks the end of a variable-length thing. For example, for a string, the delimiter would be a character or sequence of characters that do not occur in the string, and terminate it when they occur. Control information in a transmission medium (such as the record separator defined for

the ARPANET data transfer protocol) is another case of a delimiter. To indicate that the delimiter is a sequence of characters, like double dollar sign, D is assigned an appropriate value: D ← '$$'. When the delimiter is a medium-dependent control signal, D is set to a system-defined symbol, like ##EOR. A specification of such system symbols will appear in a later datalanguage spec.

Length length (LL) defines the length of a length or count field that appears explicitly in the data. Consider the LOCATION field, that varies in length from OBS to OBS. The length of this field must be indicated in each OBS, otherwise the OBS cannot be parsed. The two most straightforward ways are to indicate the count of characters in the field, or to mark the end of the field with a delimiter. When the former technique is employed, the LL parameter is used to indicate the length of the count field, in bits. Starting the item with a count, rather than ending it with a delimiter, is taken as the standard method for handling variable length items. Thus if something is variable length, and no delimiter is indicated, a count is assumed. If LL is not present, its default is used. The count immediately precedes the data itself:

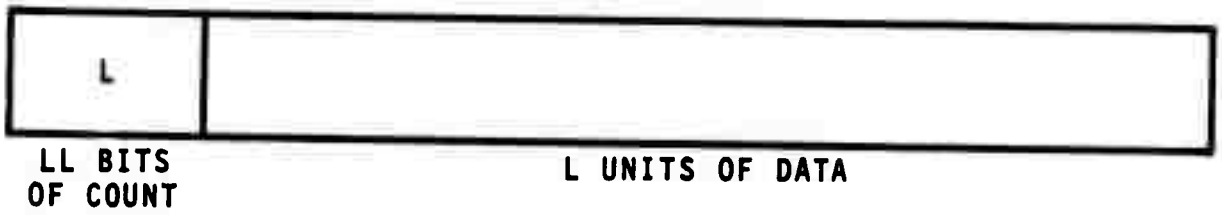


Figure 4.1

Identifier (ID) defines a bit string to appear before a self-identifying object (such as one of the elements of a mixture). An example:

```
A,MIX,D ← EOR
A1,STRING,L ← 10,ID ← B'001'
A2,STRING,L ← 15,ID ← B'010'
...
A7,STRING,LMAX ← 25,ID ← B'111'
END A
```

A is a mixture of containers named A1, A2, ... A7. They will appear in random order in A. To parse A into its components, it is necessary to associate each component with one of the containers A1, ... A7. This is done by examining the first 3 bits of each container value, which will have one of the specified identifiers. A particular occurrence of A might look like:

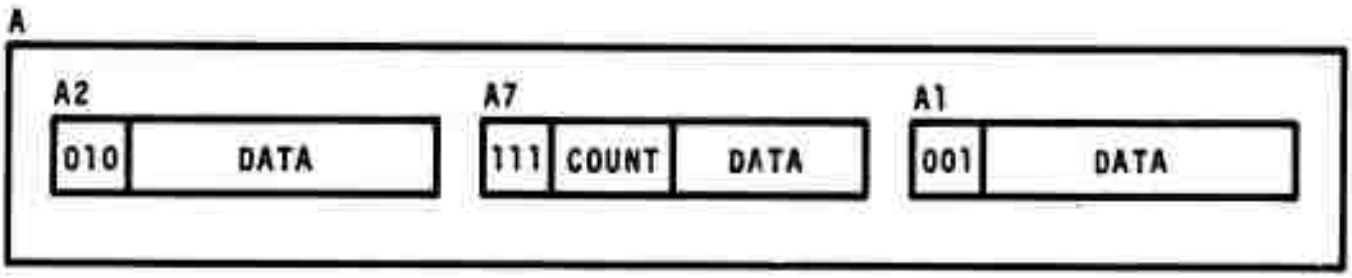


Figure 4.2

Identifiers are also used in structures with optional tails. The optional tail of a structure is a restricted form of the mixture. Components always occur in the same order (as they do in the fixed part of a structure), if they occur. They have the option of not occurring at all. They may not occur more than once.

Subscript (S) specifies the number by which the elements of an aggregate are to be known in subscripting. Normally, the first element of an aggregate is element 0, the second is element 1, etc., as in figure 4.3.

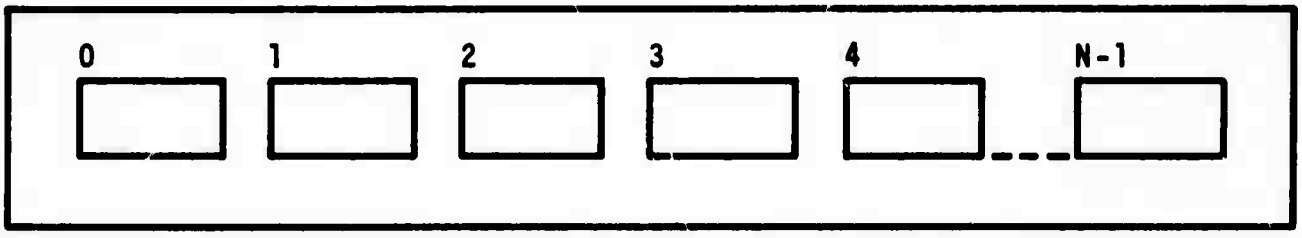


Figure 4.3

By setting S, the normal assignment of element numbers to elements is overridden:

```
A,ARRAY,L ← 20,S ← (0:100:5)
```

The 20 elements of array A will be known as element 0,5,10,15...95 instead of 0,1,2,...19. Thus A(15) is a reference to the third element of A. The S parameter does not affect the structure of an aggregate. It only changes the number by which its elements are to be referenced.

Since subscripting can be applied to aggregates of any type, the S parameter can be set for aggregates of any type.

Present (P) defines a function to be evaluated when parsing a structure, to determine if a certain component will be present.

An example:

```
A,STRUCT
A1,STRING,L ← 1
A2,STRING,L ← 10,P ← A1 = 'C'
END
```

The container A2 is present in the structure only if A1 has the value 'C'.

Real type (RT) specifies the type of floating point number when datatype is real. RT can be set to any of a set of system-defined (not user-defined) real number types. There will be one for each floating point number format in use in the network.

Character set (CS) specifies the name of a user-defined or system-defined table that specifies a character set. The CS parameter is normally set with the form STRING(cs), or simply writing cs as the datatype. It is, however, a parameter, and can be set like all others. The default for CS is ASCII.

The parameters that affect the performance characteristics of a file structure are:

Key (K) specifies the function on which a list is to be ordered. This parameter is used to set up sorted files (for example a library catalog sorted by author's last name), as well as short sorted lists. Several additional parameters, to be specified during implementation of the language, will govern the use of techniques to imitate the ordered list in cases where maintaining the physical sequence is not practical.

Inversion options (I) specify the type of information to be maintained in an inversion of the file. The presence or absence of information in the inversion affects the speed with which expressions like the following can be evaluated:

```
OBS WITH TEMP = 80 AND LOCATION = 'BOSTON'
```

It does not affect whether such an expression can be evaluated.

On the datacomputer, one list or array per file can be inverted with respect to its contents. Consider the file defined by:

```
TESTS,LIST
TRIAL,STRUCT
MATL,STRING
STRENGTH,ARRAY
PSI,INT
COMMENT,STRING,ID ← 'C'
END TRIAL
```

The TESTS file is a list of TRIALs, and the list TESTS can be inverted. Each TRIAL has two required components:

MATL - the name of the material tested
STRENGTH - an array of test results. Each result is named PSI.

In addition, a comment may appear in the TRIAL structure.

Thus, TESTS looks like:

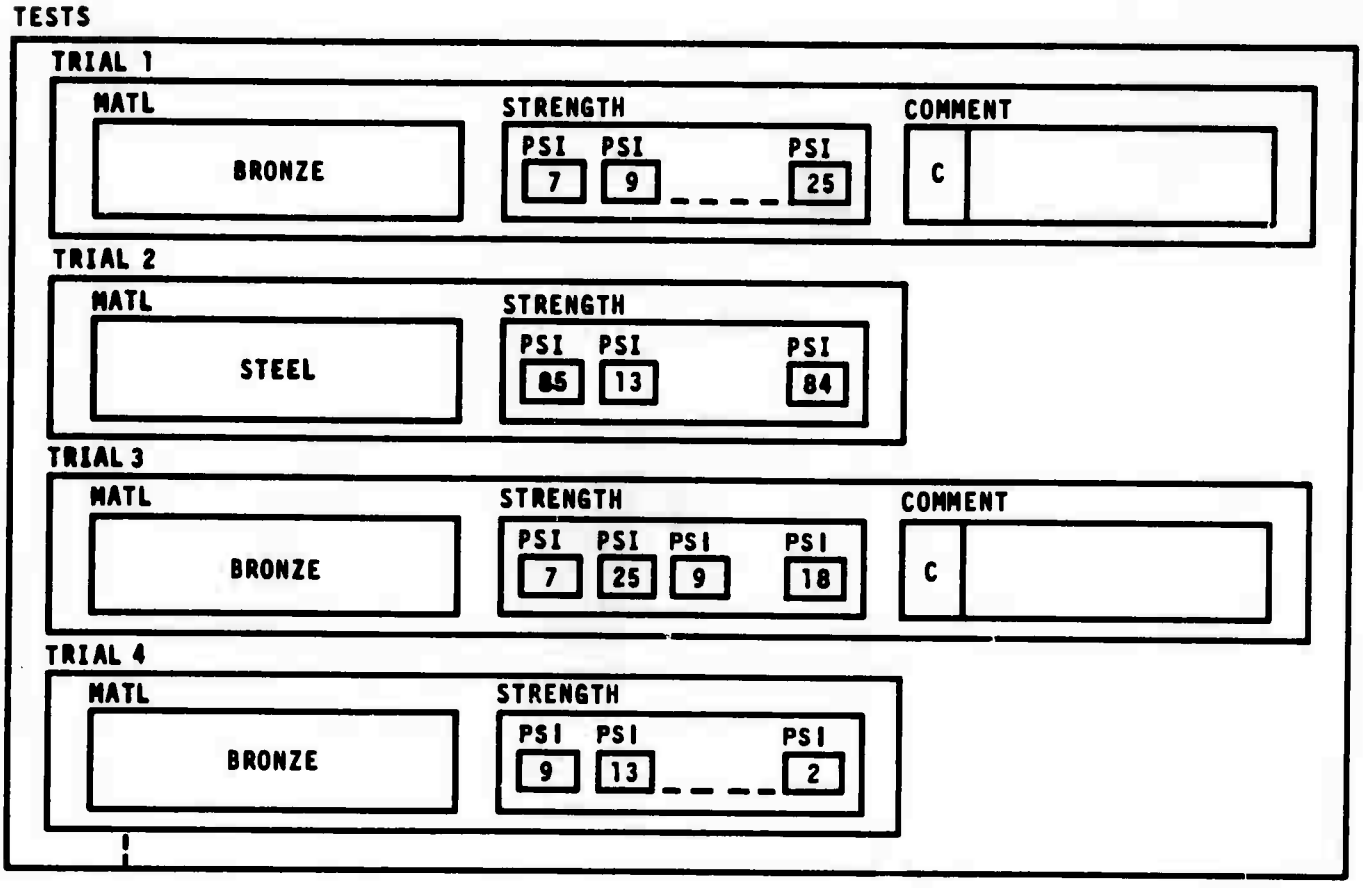


Figure 4.4

In order to invert the list TESTS, it is necessary to assign each list element (TRIAL) a permanent identifier. We have done this in the figure 4.4. The identifier is circled. An inversion of TESTS with respect to MATL is:

```
MATL = 'BRONZE' : 1,3,4...
MATL = 'STEEL'  : 2...
```

It is obvious how the maintenance of such a structure can expedite the evaluation of datalanguage expressions. The important question is how much information is worth maintaining for each container in TRIAL. Above we have indicated one option in the example for MATL. There is an entry in the inversion for each value of MATL. This is the V option.

Another possibility can be applied to optional containers like COMMENT. This is to maintain a list of the TRIALs in which COMMENT occurs:

```
COMMENT: 1,3...
```

This is the E option (exists). The E and V options are independent.

While the E option optimizes the evaluation of expressions like:

```
TRIAL WITH COMMENT
```

The V option optimizes evaluation of:

```
TRIAL WITH COMMENT = 'FOO'.
```

A third possibility applies to the PSI containers. Entries can be made in the inversion for each PSI in each TRIAL. If this is done, there are two kinds of lists that might be maintained for the PSIs. The first kind is:

```
PSI(0) = 7 : 1,3...
PSI(0) = 85 : 2...
PSI(0) = 9 : 4...
```

```
PSI(1) = 9 : 1...
PSI(1) = 25 : 3...
PSI(1) = 13 : 2,4...
PSI(2) = 9 : 3...
ETC.
```

The lists for PSI(0) are distinct from those for PSI(1), which are distinct from those for PSI(2), etc. This is the D option.

Another alternative is to keep one set of lists for all PSIs:

```
PSI = 7 : 1,3...
PSI = 9 : 1,3,4...
PSI = 25 : 1,3...
```

This is the I (indistinct) option. I, D, E, and V are independent options.

If the I option is used, it is easier to evaluate

```
TRIAL WITH PSI = 75
```

and harder to evaluate

```
TRIAL WITH PSI (7) = 75
```

than if the D option is used.

Finally there is the option of keeping information for performing range retrievals using only the inversion. That is, expressions like:

```
OBS WITH TEMP > 80
```

can be evaluated entirely on the inversion when this option is selected. The option is R (range).

To complete the example, a good choice of options for inversion of TESTS might be:

```
MATL,STRING,I ← 'V'
STRENGTH,ARRAY
PSI,INT,I ← 'VD'
COMMENT,STRING,I ← 'E'
```

Allocation parameters will be spelled out during implementation. They will allow the statement of the expected amount of space for variable length data, reserving space for the growth of structures, etc. Allocations in physical, as well as logical units will be permitted.

Additional data description parameters will be created as the need for them is recognized. Some recognized needs for which the parameters have not been designed are: alignment of containers on word/byte boundaries, choices for truncation and padding, and right/left justification.

4.6 Keyword Statements

Datalanguage has a small set of keyword statements for the specification of utility, implementation-dependent, and syntactically odd functions. Basically, keyword statements are used whenever there is no easy way or good reason to use something that is better integrated with the rest of the language (like operators or built-in functions). The list and capability of keyword statements is something that will be of great interest later on, but at present

is somewhat peripheral. The reason is that keyword capabilities are modular and quite separate from other language issues. Following is a list of the keyword statements known to be important. The list can be expected to grow:

File System Keywords

CREATE filename,parameters	creates files
DELETE filename,parameters	deletes files
PARMS filename,parameters	changes file parameters
OPEN filename,parameters	makes data and data description available for use

Data Stream Handling

PORT portname, external name, parameters	establishes data path, names it, and describes it
TRANS portname, expression	causes the set of containers referenced in expression to be transmitted through the port
CLOSE portname, expression	terminates activity on the data path associated with the port

Set Handling

DEFINE setname,expression	associates a name with the set of containers to which the the expression evaluates. Does not cause formation of the set (i.e., evaluation of the expression).

FORM setname, action	causes formation of the set according to action. The language for action depends on implementation. A good example is the action I, which limits evaluation activity to that which can be done only using the inversion.
GET setname,reference name	makes the next member of the set available for processing, and associates with it the reference name.

Compiler Control

CODE cs,mnemonic,conversion radix	defines a character set to the datalanguage compiler. cs is the name of a container that the compiler will recognize, and can be used in data descriptions following the CODE statement. The mnemonic is a single letter that can be used to write constants in expressions. Conversion radix governs the conversion of strings to numbers.
RADIX mnemonic,radix,length	defines a mnemonic and associated radix and length for integer constants appearing in expressions.

Miscellaneous

GOTO statement label	causes the interpreter to continue activity at the statement so labelled.

An example of a cost indicator is the number of OBSs that will satisfy the expression above. If they were to be transmitted to a remote program, it might pay to count the OBSs before the transmission process. The datalanguage mechanism for this problem is the COUNT function:

```
COUNT(OBS WITH TEMP > 80).
```

This function returns a single data container whose value is the count of members in the set. When the expression can be evaluated entirely on the inversion, this is a fruitful approach, since the COUNT function requires a relatively small number of accesses to the inversion to do its work. So a good indicator of transmission cost is the number of containers to be transmitted (when container size varies within known limits) and this is available through the COUNT function.

When the cost of retrieving the containers is potentially large, it is useful to get a count of the number of containers that have to be searched to evaluate the expression. This is provided with the COUNTS function, which uses information that has been extracted from the inversion by previous FORM ...,I statements. Consider the expression:

```
OBS WITH TEMP = 80 AND LOCATION = 'NYC'
```

Just for the current discussion, suppose that LOCATION is inverted and TEMP is not. Then it is trivial to find the set of OBS that have LOCATION = 'NYC' -- in fact this can be done with a few accesses to the inversion. However, to find the set of OBS that have TEMP = 80 requires a search of the OBS. The datacomputer

evaluates this expression by first consulting the inversion to minimize the number of records to be searched. Those observations that have LOCATION = 'NYC' are located from the inversion. They are searched and the ones without TEMP = 80 are discarded. The cost of the search can be anticipated by using the COUNTS function. The COUNTS function always gives the best answer it can based on the information that is available to it at the time its invoked. Normally, it is used after a FORM statement:

```
DEFINE S,OBS WITH LOCATION = 'NYC' AND TEMP = 80
FORM S,I
TRANS X,COUNTS(S)
```

Here, the COUNTS function returns through port X the number of OBSs that must be searched to evaluate expression S. In this case, that is the number of OBSs with LOCATION = 'NYC'. Were the COUNTS function invoked immediately after the definition of S, (i.e., before the FORM statement) it would say that the entire set of OBSs had to be searched (since no information has yet been extracted from the inversion).

Another useful indicator is provided by COUNTK, which counts the number of containers known to be in a set at a certain time. It is applied in the following situation:

```
DEFINE S,OBS WITH (LOCATION = 'NYC' AND TEMP = 80)
      OR LOCATION = 'BOSTON'
FORM S,I
TRANS X,COUNTK(S)
```

When S is evaluated, it will contain all the OBSs from Boston and a subset of the OBSs from New York City. At the time COUNTK is invoked above, the OBSs from Boston are certain members of S, and COUNTK will count them. Those from New York are not certain, and they will not be counted by COUNTK (the number of OBSs from New York could be determined with COUNTS).

To summarize, an effort is being made to identify and build in functions that give useful, cheap indicators of the amount of effort expected to be expended in evaluating an expression. As the important indicators are defined during this implementation, they will be added to the language as built-in functions.

5.2 Operations on Aggregate Containers

A conspicuous feature of datalanguage is that most operators are defined for most datatypes. Structures can be compared and assigned to other structures or mixtures, and even to arrays and lists. The general idea of such operations is that correspondences are established between the elements of the aggregates, and the operation is performed on corresponding elements. Unfortunately it is impossible to choose a correspondence scheme that is optimal for every application. Our approach is to choose a simple scheme likely to be useful, and provide other mechanisms for the cases it won't handle. Two other mechanisms are present now:

1. the operations can be individually indicated, element by element

2. a function can be written that makes the correspondence desired.

The automatic correspondence mechanism is defined for pairs of aggregates. It operates one way when at least one member of the pair is an array or list, and it operates another way when this condition does not hold. When an array or list is present, containers with the same subscript correspond. When no array or list is present, containers with the same name correspond. Correspondence is established recursively, one "level" of containment at a time. When there is many-to-one correspondence, or uneven many-many correspondence, a choice of action is made, based on datatype, level and operation. The heuristics used in this choice will be spelled out in a later paper.

5.3 Set Correspondences

Consider the expression:

```
OBS WITH TEMP > 80 AND LOCATION = 'NYC'
```

This expression references the set of OBS that show temperatures in New York City above 80. The compiler transforms this to

```
(OBS WITH TEMP > 80) AND (OBS WITH LOCATION = 'NYC')
```

This operation is represented by the following diagram:

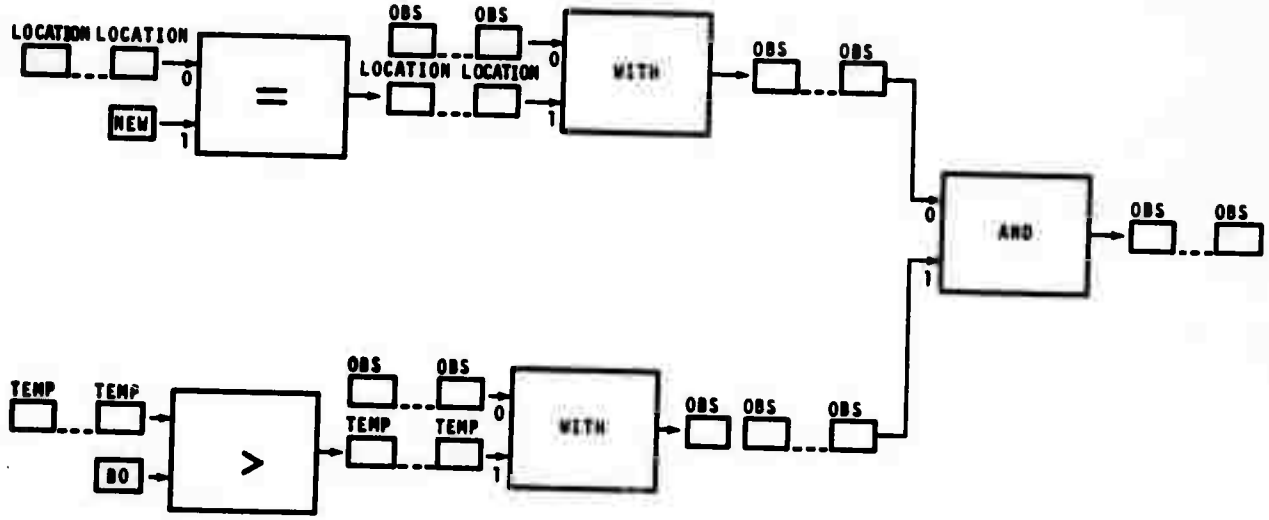


Figure 5.1

Which, for this discussion, can be simplified to:

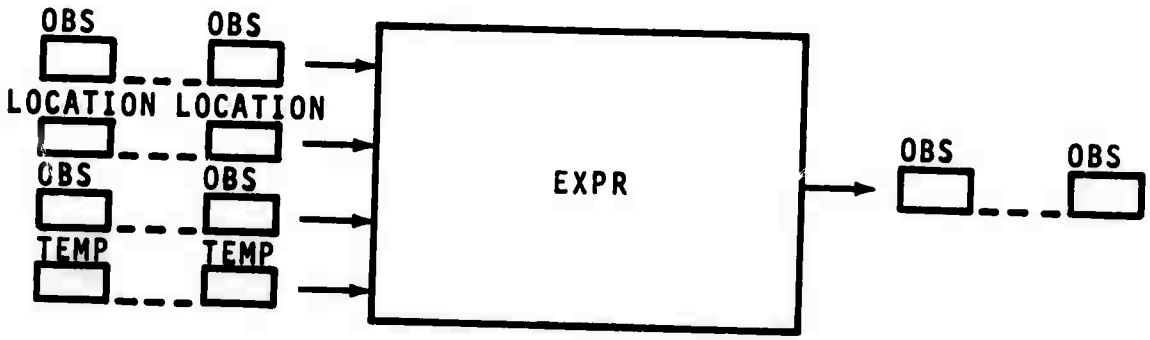


Figure 5.2

At each cycle, the function takes four containers as input:

Figure 5.3

Its output for the cycle is either 0 or 1 containers of type OBS.

Each of the input containers is drawn from a well-defined set of containers. The question unanswered so far is: How is the next container from each set chosen, and how is the choice of containers coordinated? The answer given previously is that corresponding containers are input at each port at each cycle.

To understand correspondence, it is necessary first to consider the process by which the function of four sets will be evaluated. One of the four sets will be chosen as the control set. All sets have an intrinsic order, and the intrinsic order of the control set will determine the order in which its containers are input to the function. Now the problem of selecting the other three inputs reduces to: What element in each of the input sets corresponds most

closely to the current element of the control set? For the function we are considering, this is a trivial problem. The four input sets are:

port 0: the set of all OBSs
port 1: the set of all LOCATIONs
port 2: the set of all OBSs
port 3: the set of all TEMPs.

We choose the set of all OBSs as the control set. We take the first OBS as the first input through port 0. This is also the first input through port 2. The LOCATION corresponding to this OBS should obviously be the LOCATION it contains. This solves the problem of what to input first at port 1. We make a similar choice for port 3. The process is repeated until the set of all OBSs is exhausted. Since every OBS has a TEMP and a LOCATION, all input sets are exhausted simultaneously.

This case seems to imply that corresponding means from the same container. For many datalanguage expressions, this is an adequate definition.

To find the corresponding elements of n sets, $S_1, S_2, \ldots, S_n$, find the set S' of the smallest containers C, such that every member of the n sets either is a C or is contained in a C. Then there is one group of corresponding elements for each C, and this group is made up of the elements from the n sets that either are the C or are one of its components.

This definition breaks down when one of the n sets has members that are not contained in any set of C's that also contains the

members of the other sets. The happy escape from this problem is that no particular correspondence is normally implied in expressions that give rise to such situations. Consider:

```
X ← OBS WITH TEMP > 80
```

The inputs to the function defined by this expression are:

a. the set of all X.OBS
b. the set of all WEATHER.OBS
c. the set of all WEATHER.OBS.TEMP

Here, X is a port and WEATHER is a file. The correspondence between WEATHER.OBS and WEATHER.OBS.TEMP is obvious, since the former contains the latter. However, there is no apparent way of making correspondences between X.OBS and WEATHER.OBSs. But this is not a problem. The expression really means: evaluate OBS WITH TEMP > 80 and assign each output OBS to some X.OBS. A good rule for this situation is -- make any correspondence that doesn't waste X.OBSs. This rule is suitable for all the cases in which the first rule breaks down (that the designer currently knows about). No doubt a more elegant way to state this second rule will be found in time. For now, let it suffice that there is an obvious and simple way to handle the exceptions to the first rule.

Finally, consider a case when the first rule applies, but appears to supply an inadequate solution.

```
A, LIST
B, STRUCT
B1, LIST
```

```
B1A, STRING
END B
B WITH B1A = 'X'
```

Each B contains many B1As. Thus, one group of inputs to the function is:

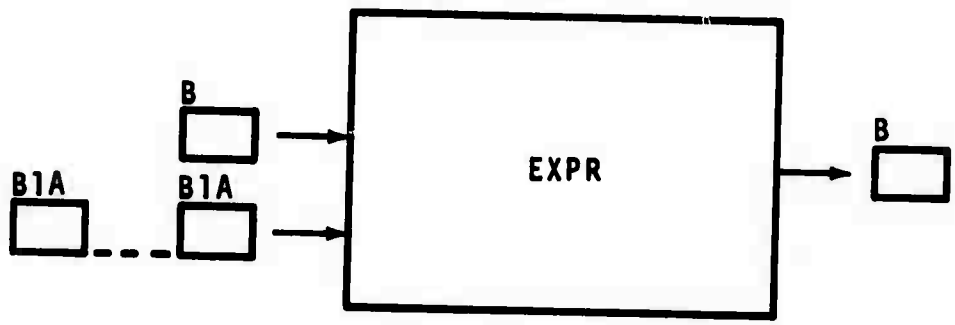


Figure 5.4

All the B1As in a single B1 correspond to the same B.

There are a number of choices:

a. Choose a particular B1A and throw the rest away. Now we have the familiar situation where there is one container at each input port.

b. Rewrite the function so this doesn't happen.

c. Do something more complicated.

For the moment, we prefer (b). We'll think about (c) for a while and see if anything promising appears. If ever (b) fails, we'll fall back on (a) so that all syntactic expressions are defined. In this case, we rewrite the expression as

```
B WITH B1(0) = 'X'
OR
B WITH B1(1) = 'X'
OR
:
B WITH B1(n) = 'X'
```

which makes the function more agreeable:

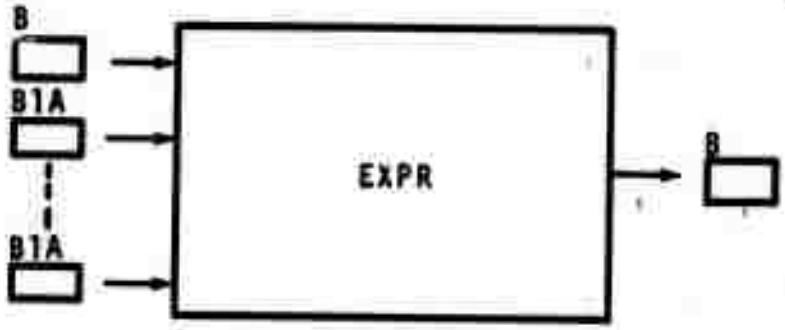


Figure 5.5

The work on correspondences is not yet complete. The intention is to restrict the correspondence mechanisms to simple-minded techniques. This limits the power of the datalanguage expression,

but it's not obvious that this limitation excludes the use of the expression in any particularly valuable areas. Further, a general mechanism (the FOR loop) in the datalanguage is capable of handling any situation to which the expression cannot be applied.

Timings and Storage Estimates

Datacomputer Project
Working Paper No. 4
November 15, 1971

Contract No. DAHC04-71-C-0011
ARPA Order 1731

Computer Corporation of America
575 Technology Square
Cambridge, Massachusetts 02139

Preface

The present document discusses the storage space required for datacomputer files and the time required to execute requests on the datacomputer.

Other documents--issued and to be issued in the present series--discuss the software architecture of the datacomputer, the hardware configuration, datalanguage, and related topics. All documents in the series are working papers, and subject to revision without notice.

Table of Contents

Chapter 1
Introduction

1.1 Overview of the Datacomputer

The datacomputer is a system which performs data storage and data management functions.

One may consider the datacomputer to be a black box with multiple physical ports to which processors can be interfaced. (See Figure 1.1.)

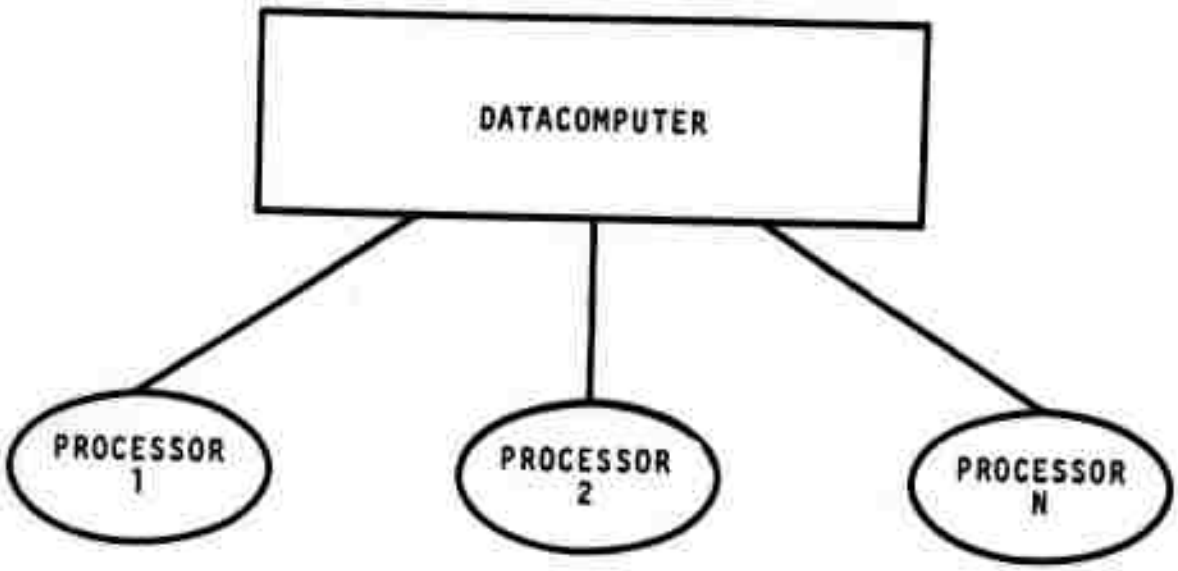


Figure 1.1

Each of the processors can itself have multiple users, which can avail themselves of the data storage and data management services that the black box offers.

The datacomputer offers the following types of service:

1. On-line storage of files and file descriptions. These files can be extremely large, with an upper limit of 10^{12} bits.

2. Retrieval of data (whole files, subsets of files, individual data elements).

3. File maintenance functions, that is, addition of new data, deletion of old data, changes to existing data.

4. Data reformatting.

5. Backup and recovery mechanisms.

6. Accounting.

7. Data security, preventing users from getting unauthorized access to data.

8. Data sharing, allowing multiple users to access the same data bases.

9. Simultaneous multi-user access, allowing multiple requests to be operated on simultaneously.

Interaction with the datacomputer is through a specialized system of notation called datalanguage. Within datalanguage, one can express all requests for data storage and data management services of which the datacomputer is capable. When datalanguage statements

are presented at the ports of the datacomputer, the system proceeds to perform the desired service request. Datalanguage statements can originate in a user program directly, in the compilation process of a user program, or in the operating system of the external computers.

The datacomputer for the ARPANET is physically connected to an IMP and the Illiac IV, as shown below:

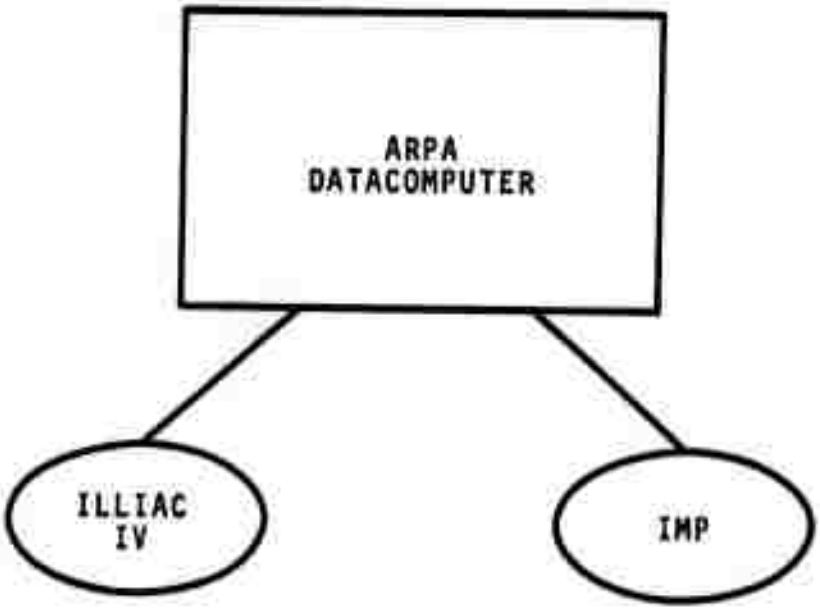


Figure 1.2

The datacomputer sees all users as remote programs connected to it via one or more one-way logical paths. Each using program establishes one path for the communication of datalanguage to the datacomputer; additional paths may be established for the transfer of data in either direction.

1.2 Purpose of this Paper

This paper discusses estimates for storage space and access times for files stored on the ARPA datacomputer. It is based on the current specifications for the Unicon 690 and for the internal file structures. Since these specifications are subject to change, and since the estimating process is crude, the estimates in this paper are the best guesses possible at this time. Wherever possible, the conditions under which these estimates are expected to be valid have been identified.

Chapter 2
Space Estimates

2.1 Factors Affecting File Size

The amount of storage space required for a file stored on the datacomputer depends on a number of factors. These include: the amount of data in the file, the amount of updating to be done, whether or not the file is sorted, and how much of the file is inverted.* In order to estimate the storage space required for a file, the file sponsor must indicate approximate values of some of the file's properties.

This chapter isolates what are expected to be the most significant factors in determining file size. The rest of this section discusses what these factors are. The following sections give formulas for estimating file size. The space required for the inversion lists and the space required for the rest of the file are estimated independently. Total file size is found by adding the two figures. Section 2.2 discusses the unsorted files and Section 2.3 discusses sorted files. Space for the inversion is discussed in Section 2.4.

Record Formats.

The data description for the file, which the file sponsor develops in datalanguage, specifies the record format. In the terms used in datalanguage, the file is normally a list and the data containers in the list are records.** The length of a particular record is the sum of the lengths of each of the data containers which it contains. The length of a simple data container is the length of the value plus the length of the identifier plus the length of either the length field or the delimiter. If the data container is fixed-length, then there is neither a length nor a delimiter.

---

* See Working Paper #3, Datalanguage.

** There are other possibilities. These will be discussed in future working papers.

If the data container is required, then no identifier is necessary.

To compute the space required for the file, the user must estimate the average length of records in the file. If record lengths vary greatly from the average, then there is slightly more overhead than indicated in this chapter.

In addition to the space required for the record itself, there is system overhead for variable-length records. Exactly how much overhead depends on the update characteristics of the file, but it is a minimum of 16 bits per record in non-updatable and append-only files and 32 bits per record in updatable files. This overhead is not included in the estimate of the average record length.

Sorted Files.

A file is sorted if the parameter KEY is set in the data description. The datacomputer maintains additional tables to permit fast retrievals based on the values of the sort keys. Except for files with extremely small records, these tables are insignificant relative to the size of the rest of the file. Also, new records that are added anywhere in the sort sequence take up more space (slightly more than 17 bits per record) than records that are added at the end of the file. Section 2.3 gives a more accurate estimation of how much space a sorted file requires.

Updatability.

There are two different kinds of updating -- changing existing records and adding new records. The file sponsor must estimate the number of records to be added to the file. As discussed above, in a sorted file there is approximately 17 bits additional overhead

for records that are added after the file is initialized. For unsorted files the overhead for additional records is negligible.

The file sponsor must also determine if updating will change the record length of existing records. Of course, if records are fixed-length this will not happen. The most significant kind of record changes to consider when allocating space is the addition of optional data containers to existing records. For example, in a hospital file containing one record for each patient, the number of occurrences of a data container named MEDICATION is variable and can be changed in existing records.

In the sections below, the files are divided into four categories based on the amount of updating expected: non-updatable, append-only, moderately updatable, and volatile. (An append-only file is one in which records may be added to the end of the file but not between existing records.) With the exception of append-only files, these categories are based on the total amount of updating to be done, no matter what kind. Whether a file is to be considered moderately updatable or volatile depends on how much updating is to be done between garbage collections. These classes reflect slightly different means of using the storage medium based on the amount of updating expected.

Inversion.

A data description can specify inversion options for a file. The datacomputer maintains lists of records for each value that occurs in inverted data containers. The file sponsor must estimate how much space these lists require. The procedure for doing this is discussed in Section 2.4.

2.2 Unsorted Files

The amount of space required for system overhead is minimal for unsorted files. The size of the inversion lists is discussed in Section 2.4. The other factors to be considered are the size and number of records and the amount of space left for updating.

The user must estimate values for the following variables:

N--the number of records in the file originally
A--the number of records to be added to the file
L--the average record length, in bits
q--the fraction of L by which existing records are expected to grow.

The size of the file can be approximated as

$$(N+A)(L).$$

The following chart gives a more accurate set of formulas, which take into account whether records are fixed- or variable-length and in what update class the file falls. Note that for unsorted files the update class is based only on how much records are changed and not on how many records are added.

	Fixed-Length Records	Variable-Length Records
Non Updatable or Append Only	(N+A)(L)	(N+A)(L+16)
Moderate Updating	1.13(N+A)(L)	1.13(N+A)(L+qL+32)
Volatile	1.5(N+A)(L)	1.5(N+A)(L+qL+32)

Note that in the chart above A=0 if no new records are ever added and q=0 if old records are never changed.

As an example, consider a file containing a million records with an average record length of 200 8-bit bytes. If the file is not updatable and the records are fixed-length, then the space required is

$$(1{,}000{,}000)(200*8)$$
$$=1{,}600{,}000{,}000 \text{ bits.}$$

If the records are variable-length and a lot of updating is expected then the formula in the last row and the last column is used. Assuming that updating does not change the average record length but leaving room for the expansion of some of the existing records, set q=1/10. Allow the number of records to grow by 20%. Then the space required for the file is

$$1.5(1{,}000{,}000+200{,}000)*(1600+1600/10+32)$$
$$= 3{,}225{,}600{,}000$$
$$= 3.2 \times 10^9$$

2.3 Sorted Files

In sorted files space is left for inserting records and for additional tables maintained by the system. The variables used to estimate size of the basic file are the same as the ones used in unsorted files. As with unsorted files, (N+A)(L) is usually a good estimate. However, the chart below gives more accurate formulas for determining the number of bits in the body of the file, assuming that the new records are distributed evenly throughout the file. These formulas should handle the special cases for which system overhead is not negligible. (For append-only files, the formulas for unsorted files are more accurate.)

	Fixed-Length Records	Variable-Length Records
Non-updatable	(N)(L)+(N)(L)(48)/32000	N(L+16)+N(L+16)(48)/32000
Moderate Updating	1.13[(N+A)(L+17)+.01 (L+64)]+48NL/32000	1.13[(N+A)(L+qL+49)+.01 (A)(L+qL+96)]+48(N) (L+32)/32000
Volatile	1.5[(N+A)(L+17)+.01(A) (L+64)]+48NL/32000	1.5[(N+A)(L+qL+49)+.01(A) (L+qL+96)]+48(N)(L+32)/ 32000

Note that the last term in the above formulas is insignificant except in very large files.

Consider a file with a million records with an average record length of 200 bytes. Allowing for change to existing records and the addition of 20% more records, set q=1/10, A=200,000, and f=1.5. Then the number of bits if the file is sorted is

$$
\begin{aligned}
&1.5[(1{,}000{,}000+200{,}000)(1600+160+49)\\
&\quad +.01(200{,}000)(1600+160+96)\\
&\quad +48(1{,}000{,}000)(1632)/3200]\\
&\quad = 3{,}284{,}400{,}000\\
&\quad = 3.3 \times 10^{9}
\end{aligned}
$$

Note that this is 5.88×10^{7} bits or less than 2% more than the same file if it is not sorted. (See Section 2.2.)

2.4 Inversion Lists

Inversion lists are stored in different formats. The main factor in determining the storage format for a particular list is the length of the list relative to the size of the file. In order to estimate

the size of the inversion the user must estimate the size as well as the number of the lists. To do this, he must guess how many values exist for each inverted data container and in how many records each value occurs.

Values that occur in inverted data containers are divided into three classes. Class 1 are the values that occur exactly once in the file. Class 2 are the values that occur in approximately 6% of the records of the file. (For volatile files, the percentage is slightly higher. For files that are not updated much at all, the percentage is slightly lower.) Class 3 are the values that occur in more than 6% of the file.

For example, consider a personnel file with a record for each person. Each data container named SOCIAL SECURITY NUMBER has a unique value. All of the values for SOCIAL SECURITY NUMBER are in Class 1. If less than 6% of the people receive a particular salary, then that value of the SALARY data container is a Class 2 value. (Note that some salaries might be Class 1 or Class 3, but most are likely to be Class 2.) The values for the data containers named SEX are MALE and FEMALE; these two values occur frequently in the file and are in Class 3.

Data containers for which the inversion option range (R) is specified are considered to have more values than actually occur in the file. This allows for evaluation of datalanguage expressions like AGE>25. The number of additional values is

10*(maximum number of significant digits)+2

These additional values are all considered to be in Class 3. (For files with a large number of records this results in an estimate which is high.)

If the same value occurs in two data containers with different names, it is considered to be two values. Likewise, if the option D (distinct) is assigned to a data container, then a value must be counted separately for each subscript with which it occurs.*

A file with less than 2,000 records is not considered to have any values in Class 3 at all.

To estimate the space required for the inversion, the following variables must be estimated:

C_1 -- the number of values in Class 1
C_2 -- the number of values in Class 2
C_3 -- the number of values in Class 3
N -- the number of records at file initialization time
A -- the number of records to be added after file initialization time
N_2 -- the average number of original records in which a particular value in Class 2 occurs
A_2 -- the average number of new records in which a particular value in Class 3 occurs.

The file must be classified according to how much updating is to be done. For the purpose of estimating inversion size it does not matter whether new records are added at the end of the file (i.e., in append-only mode) or in sort order. Updatability depends on how many records are to be changed and how many are to be added, relative to file size, between garbage collections. A file is classified as non-updatable, moderately updatable, or volatile. Let f be set to

1 -- for non-updatable files

---

* See Working Paper #3, Datalanguage.

1.13 -- for moderately updatable files
1.5 -- for volatile files

Then for a file with less than 32,000 records, the size of the inversion in bits is

$$(f)*[70C_1+C_2(86+16N_2+32A_2) + 2^{15}C_3+2^{15}]$$

Letting S=(N+A)/32,000, rounded up to the nearest integer, the number of bits in the inversion if there are more than 32,000 records in the file is

$$(f)*[70C_1+C_2(70+48S+16N_2+32A_2)+C_3(70+32S(2^{10}+1))+2^{15}S]$$

A few things should be noted about this formula.

1. A value that occurs in only one record in the file takes up about 70 bits in the inversion.

2. The space required for values in Class 2 depends on both file size and the number of records in which they occur. Less space per record is required for original records than for records that have been added to the file.

3. The space required for values in Class 3 is independent of the number of records in which they occur. Thus, there is virtually no cost in inversion space associated with adding a Class 3 value to a record.

Take a sample file of 100,000 records. The value of S is 4. If the only inverted data container is a unique identifier, which occurs in every record, then the size of the inversion is approximately

$$(70)*(100,000)+4*2^{15}$$
$$= 7,131,072$$

This is likely to be insignificant compared to the size of the file.

Consider inverting two more data containers, one with 25,000 values distributed evenly throughout the records of the file and one with 2 values, each occurring in half of the records in the file. The values in the first data container are in Class 2 with $N_2=4$. The values in the second data container are in Class 3. Then the size of the inversion for the original file, without space for updating, is

$$(70)(100,000)+(25,000)(70+48*4+16*4)+2[70+(32)*(4)(2^{10}+1)]+2^{15}*4$$
$$= 15,543,612$$
$$= 1.5 \times 10^7 \text{ bits}$$

Allowing for 20% more records and changing existing records let $S=4$, $A_2=.8$ and $f=1.50$. Then the size of the inversion is approximately

$$1.50[(70)(100,000)+(25,000)(70+4*48+4*16+.8*32)+2(70+32*4(2^{10}+1)+4*2^{15}]$$
$$= 26,864,795$$
$$\sim 2.7 \times 10^7 \text{ bits}$$

To take one more example, look at how inversion size is affected by the range (R) option. In the same sample file of 100,000 records, consider another data container with numerical values ranging between 1 and 100. Any particular value is likely to occur in about 1000 records or 1% of the file. That adds 100 values in Class 2. Allowing three significant digits, it also adds 32 Class 3 values. This increases the size of the inversion for the original file by almost 2×10^7 bits. Note that while this is somewhat costly in space, for certain applications the difference in retrieval times is even more significant.

Chapter 3
Timings

3.1 Factors Affecting Retrieval Time

The time required to execute a datalanguage request depends on such factors as: the mass storage device used, the number of requests being serviced by the datacomputer, the kind of service requested, and the update characteristics of the file.

In this chapter timing estimates are considered for one user only, not for a time-sharing situation. The rest of this section discusses what are expected to be the most significant factors and why. The following sections give estimates of timings under varying conditions.

The Mass Storage Device.

The mass storage device used in the datacomputer is the most significant determinant of the time required to execute datalanguage requests. In the datacomputer for the ARPANET, Precision Instrument's Unicon 690 is the device being used. Because of the nature of the device, it takes between five and ten seconds after a user opens a file before data can be read from or written to the file. Once a file is positioned for reading the amount of time it takes to get at a particular piece of data varies from 0 to about 500 ms. with an average of about 375 ms. Furthermore, the effective data rate depends on the amount of updating that was originally allowed for and the amount that has actually been done.

The timing estimates in this chapter are based on the assumption that the speed of the Unicon is the limiting factor in executing datalanguage requests. The data rate over the ARPANET is not included in the estimates. Requests that are compute-limited

(e.g., large sorts or requests requiring a lot of arithmetic) are not considered. The timings do not include the original 10-second delay.

Retrieval Conditions.
The datalanguage conditions that determine which data containers are in a set also determine how those data containers will be found. Take, for example, the datalanguage expression

```
PERSONS WITH AGE = 25
```

If the inversion option value (V) has been specified in the data description for the file, then the inversion can be used to find all such PERSONS records without examining any of the other records in the file. Likewise, only if the option range (R) has been specified can the expression

```
PERSONS WITH AGE > 25
```

be evaluated without sequentially searching all of the records in the file. Unlike sequential searches, the access time when using the inversion is not very sensitive to file size. However, if a data container in a retrieval condition is not inverted, then normally a sequential search of all the records in the file is the only way to find the records.

Yet another mechanism is used to evaluate expressions which specify sort ranges. Again, this affects the time it takes to execute the request.

Number and Distribution of Records Retrieved.
When using the inversion to locate records, the number of records retrieved is one of the most significant variables in determining

the amount of data to be read from the laser memory. Because access times for the laser memory can vary by as much as an order of magnitude, the distribution of the records within the file also greatly affects the time required to read them. For example, reading 10 records takes 4.2 seconds if they are scattered throughout a file and as little as 375 ms. if they are located near each other.

Update Characteristics of the File.

Retrieval times are affected by the quantity of updating expected and the quantity of updating done in a file. For one thing, the Unicon is used in slightly different ways depending on the updatability of the file. Storage formats that permit more updating also result in slower effective data rates from the Unicon.

More significant than the storage formats used is the updating that has been done relative to what was expected. As long as the amount of updating has not exceeded what was originally allowed for, retrieval times are virtually unaffected by updates. Theoretically, under most conditions there is no limit to how much file maintenance can be performed, but in practice performance degrades rapidly after the space originally allotted for updates is used up. Timings for files that have exceeded the original allocations are not considered in this working paper. It is assumed that when a file reaches that state, it is garbage collected.

Special Conditions.

The estimates in this chapter do not apply to files with certain special properties. These properties include: record lengths greater than 32,000 bits; record lengths less than 100 bits; greatly varying record lengths; total file size of less than 4×10^7 bits. Files with more than 2×10^9 bits are also special; they are discussed in Section 3.5.

3.2 Sequential Searches

One way of accessing a file is sequentially. This means that every record in the file is examined, and they are examined in the order in which they are stored. Sequential searches are used when the records in a particular set can only be identified by looking at all the records in the file. The time required to perform a sequential search of a file is proportional to file size.

Let B be the number of bits in the file without the inversion. Then a sequential search takes

$$(3.1 \times 10^{-4})(B) \text{ ms.}$$

For example, for a non-updatable file of 100,000 records with an average record length of 200 bytes or 1600 bits, file size* is 1.6×10^8 bits and the search takes

$$(3.1 \times 10^{-4})(1.6 \times 10^8) \text{ ms.}$$
$$= 49.6 \text{ seconds}$$

For a non-updatable file of 1,000,000 records with the same record length, or for a file with 100,000 records and a record length of 2000 bytes, the search takes 496 seconds or 8.3 minutes.

A variation is sequentially searching all the records in a given sort key range. On an average it takes 725 ms. to find the beginning of the sort range. Thus, letting B be the total number of bits in the records in the sort range, the time required for this kind of retrieval is

---

* The file size includes the system overhead discussed in Sections 2.2 and 2.3. For non-updatable files the number of records times the bits per record gives a good estimate of file size.

$725 + (3.1 \times 10^{-4})(B)$ ms.

Note that specifying a sort key range can substantially cut down the time required for a sequential search.

3.3 Inversion Retrievals

When all of the data containers used in a set of retrieval conditions are inverted, then the retrieval is performed in two steps. First the inversion is used to identify which records are to be retrieved. Then the records themselves are found. (Note that only the first step is necessary in order to count the records with a given set of values.) Each step will be discussed separately. In estimating the total time it takes to execute a datalanguage request, remember to add the two numbers.

Inversion Look-up.

The same classes of values are used in estimating inversion look-up time as are used in estimating the size of the inversion (see Section 2.4). The values that occur in exactly one record in the entire file are in Class 1. Class 2 contains the values that occur in less than 6% of the records in the file. The values that occur in more than 6% of the records in the file are in Class 3.

Even if the range (R) inversion option has been specified for a data container, when used with an = (e.g. AGE=21) it is treated as any other data container. Range retrievals (e.g. AGE>21) are discussed further below.

When a value in Class 1 is used to find a single record, the inversion look-up takes an average of 400 ms. Locating the record takes another 400 ms. for an average total retrieval time of 800 ms.

When looking for a number of Class 1 values, the average look-up time per value goes down. How much it goes down depends mainly

on the total number of values in the file. If there are less than 20,000 values in the file, each additional look-up (i.e., after the first) takes 70 ms. If there are more than this, then normally it takes about 350 ms. to look up each additional value.

For values in Class 2 and Class 3, the look-up time in the inversion (as well as the space required in the inversion for these values) depends on the distribution of the values throughout the file. Assume that each value is evenly distributed throughout the file. If this assumption is not true, the timing estimates given below are high. They are also high for files with less than 32,000 records.

Let

S = (#records in file/32,000) rounded up to the nearest integer (In a sorted file, the number of records is the total number for which room is allocated.)

P = # properties specified in the Boolean expression

Then to identify the records with a particular combination of properties, all of which are in Class 2 or Class 3 takes

310 + 70P + 370S*(P+1) ms.

For example, the expression

AGE=25 AND SEX=MALE

contains two properties. The inversion look-up time in evaluating this expression in a file of 100,000 records is

310 + 70*2 + 370*3*3
= 3780 ms.

In a file of 1,000,000 records the time required is

$$310 + 70*2 + 370*30*3$$
$$= 33.7 \text{ sec.}$$

For a range retrieval the number of properties is 2 + 2 (# digits). Thus, the expression

AGE>25

contains 4 properties. The time required to look up numeric properties in the inversion is

$$310 + 70P + 47SP + 445S$$

Thus, identifying the records with AGE>25 in a file of 100,000 records takes

$$310 + 70*4 + 47*3*4 + 445*4$$
$$= 2934 \text{ ms.}$$

In a file of 1,000,000 records it takes

$$310 + 70*4 + 47*40*3 + 445*40$$
$$= 21.0 \text{ seconds}$$

Locating Records.

Once a set of records has been identified from the inversion, the records themselves must be located. The time required to get the records depends on the number of records and their distribution in the file. If the records are scattered throughout the file, then the time also depends on file size. Assume that the records are distributed evenly throughout the file.

First consider a file with an average record length of 200 bytes. If more than 1% of the file is to be retrieved, then the time required to locate them is proportional to file size. (See Section 3.2.) Otherwise, the time is proportional to the number of records being located, where the factor of proportionality depends on the percentage of the file being retrieved. Let t equal

46 -- if 1% of the file is being retrieved
70 -- if between .05% and 1% is being retrieved
375 -- if less than .05% is being retrieved.

Then the time in ms. required to locate the records is t*R. Thus, to locate 1,000 records takes 46 seconds in a file of 100,000 records and 70 seconds in a file of 1,000,000 records. To read 100 records takes 7 seconds in a file of 100,000 records and 37.5 seconds in a file of 1,000,000 records.

Now consider a file with an average record length of 1000 bytes. If more than 6% of the records in a file are to be located then the time is proportional to file size. Otherwise, let t =

46 -- if 6% of the file is being read
70 -- if between .3% and 6% of the file is being read
375 -- if less than .3% of the file is being read.

Again, the time in ms. to locate R records is t*R.

3.4 Sort Key Retrievals

A retrieval request can either specify a particular sort key value or a range of sort key values. The sort key range may be the only retrieval specification, or it may be used along with other record properties.

If one unique sort key value is specified, then the retrieval time depends on whether or not the value is in the inversion. If it is, then the sort key value is treated in the same way as any other unique value. (See Section 3.3.) It takes 750 ms. to read the record. If sort key values are not in the inversion, it takes an average of 1130 ms. to find the record if the record is an original record, and between 1130 and 1875 ms. if the record is one that has been added to the file.

When the only retrieval specification is a sort key range, then the time in ms. required to read the records in the range is

$$750+(\#\text{bits in sort range})(3.1 \times 10^{-4})$$

When a retrieval specification includes both a sort key range and values that are in the inversion, retrieval is a 3-step process. In addition to the 2 steps described in Section 3.3, the records in the sort range must be identified. For non-updatable or append-only files, this takes 1265 ms. In other files it takes 2015 ms*. Remember that this is in addition to the time required for the inversion look-up and the location of the records.

3.5 Large Files

Because of the nature of the Unicon, almost 2 billion bits are accessible at a time. Files that are larger than this are divided into sections of roughly 2 billion bits. There is normally a 10-second delay for each of the sections, though in some cases this delay can be avoided. To estimate execution time for a datalanguage request, estimate the time for one section and multiply by the number of sections.

---

* If the sort key range is small, say less than 1000 records, then these estimates are high. If a lot of new records have been added, relative to what was originally allowed for, these estimates are low.

Specifying a sort key range eliminates the need to look at the entire file. It takes an insignificant amount of time to identify the sections containing a particular sort range. Thus, if the range falls within one section, the estimates in the sections above are applicable. If the range falls in more than one section, then the 10 second delay per section plus the time per request as discussed above is multiplied by the number of sections.